TURING 图灵程序设计丛书 C/C++系列

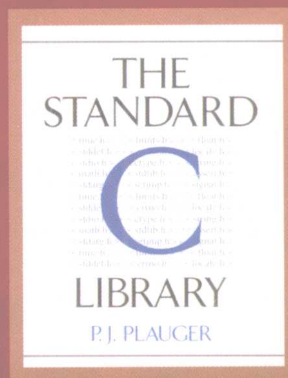
PRENTICE HALL

The Standard C Library

C标准库

[美] P.J. Plauger 著
卢红星 徐明亮 霍建同 译

- C标准库“圣经”
- 提供完整源代码，全面深入阐述库函数的实现与运用
- C程序员必备参考书





人民邮电出版社
POSTS & TELECOM PRESS

TURING 图灵程序设计丛书 C/C++系列

The Standard C Library

C标准库

[美] P.J. Plauger 著
卢红星 徐明亮 霍建同 译

人民邮电出版社
北 京

图书在版编目（CIP）数据

C 标准库 /（美）普劳格（Plauger, P.J.）著；卢红星，徐明亮，霍建同译．—北京：人民邮电出版社，2009.7
（图灵程序设计丛书）
ISBN 978-7-115-17286-0

Ⅰ. C… Ⅱ. ①普…②卢…③徐…④霍… Ⅲ. C 语言－程序设计 Ⅳ. TP312

中国版本图书馆 CIP 数据核字（2007）第 187460 号

内 容 提 要

本书集中讨论了 C 标准库，全面介绍了 ANSI/ISO C 语言标准的所有库函数。书中通过引用 ISO C 标准的相关部分，详细讲解了每一个库函数的使用方法，并通过示例描述了其实现细节，且给出了实现和测试这些函数的完整代码。此外，每章结尾附有不同难度的习题，帮助读者巩固和提高。通过此书，读者将会更好地使用 C 标准库，并学会如何设计和实现库。

本书结构清晰，内容权威，阐述精辟，对于各层次 C 程序员和相关专业高校师生都是一本优秀的参考书。

图灵程序设计丛书

C 标准库

◆ 著 [美] P. J. Plauger
译 卢红星 徐明亮 霍建同
责任编辑 傅志红 刘艳娟

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街14号
邮编 100061 电子函件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京顺义振华印刷厂印刷

◆ 开本：800×1000 1/16
印张：30.5
字数：761千字 2009年7月第1版
印数：1－3 000册 2009年7月北京第1次印刷

著作权合同登记号 图字：01-2007-0546号

ISBN 978-7-115-17286-0/TP

定价：79.00元

读者服务热线：(010)51095186 印装质量热线：(010)67129223

反盗版热线：(010)67171154

版权声明

Authorized translation from the English language edition, entitled *The Standard C Library*, 9780131315099 by P. J. Plauger, published by Pearson Education, Inc., publishing as Prentice Hall, Copyright © 1992 by P. J. Plauger.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

CHINESE SIMPLIFIED language edition published by PEARSON EDUCATION ASIA LTD. and POSTS & TELECOM PRESS Copyright © 2009.

本书中文简体字版由 Pearson Education Asia Ltd. 授权人民邮电出版社独家出版。未经出版者书面许可，不得以任何方式复制本书内容。

本书封面贴有 Pearson Education（培生教育出版集团）激光防伪标签，无标签者不得销售。

版权所有，侵权必究。

译 者 序

理论上，在学习完任何编程语言的基本语法后，我们就可以用它来编写程序以解决任何实际的问题了。但是，熟练地使用语言中已经提供的代码库有助于我们在编程时极大地减少工作量和避免不必要的错误。语言所衍生出的任何能提高生产力的库，其重要性往往会远远超过该语言本身。对这些库的了解程度，也是评判一位优秀程序员的重要标准。因此，学习一门语言并学习其库的实现，这样才能熟练地驾驭语言工具，了解库所提供的功能和局限性，进而在特定的应用要求下扩展库——这往往是初学者普遍忽视的一个重要细节。

C 语言更是如此！尽管它及它所附带的 C 标准库中隐藏着太多的晦涩陷阱，但用 C 语言编写的程序却已深入到软件行业的各个关键角落，使我们不得不要花大力气“过分”地钻研它。已经有太多的书关注了 C 语言本身及 C 标准库应如何使用，本书却独树一帜，它将告诉你 C 标准库是如何用标准 C 来实现的。学习 C 标准库本身的实现，就是学习 C 语言最好的教材，因为其实现过程将会把使用 C 语言编写具有工业强度的健壮代码所需的技巧展现得淋漓尽致。

如果想仔细探究 C 标准库的实现细节，P. J. Plauger 的这本书将是你最好的选择[①]。C 标准库由在 15 个头文件中声明的函数、类型定义和宏组成，每个头文件或多或少代表了一定范围的编程功能。有人说，标准库可以分为 3 组，如何正确和熟练地使用它们，可以相应地区分出 3 种层次的 C 程序员：

- 合格程序员，`<ctype.h>`、`<stdio.h>`、`<stdlib.h>`、`<string.h>`；
- 熟练程序员，`<assert.h>`、`<limits.h>`、`<stddef.h>`、`<time.h>`；
- 优秀程序员，`<float.h>`、`<math.h>`、`<error.h>`、`<locale.h>`、`<setjmp.h>`、`<signal.h>`、`<stdarg.h>`。

如果你反复研读本书，并能将本书所提供的 9 000 余行关键实现代码中所蕴藏的 C 语言精髓熟稔地应用到你正要或将要从事的实际开发中去，那么，毫无疑问，你已完全超越了上面“优秀程序员”的标准，成为“超优秀程序员”！

希望这本令我在翻译过程中获益匪浅的书也能给你带来美妙而独一无二的阅读享受！

最后，感谢人民邮电出版社图灵公司刘江和傅志红编辑的邀请和信任，他们踏实负责、兢

① P. J. Plauger 参与了 C 和 C++ 标准库的开发，也是 *The Standard C++ Library* 一书的作者。——译者注

兢业业的工作精神极大地鞭策了我；感谢霍建同、贾彦磊、万振杰等参与了部分章节的初译及整理工作；感谢郑州大学的卢红星老师认真审阅和复译全稿，他深厚的技术积淀、丰富的教学经验以及一丝不苟、精益求精的严谨态度也令我在翻译过程中获益良多；更要感谢我的妻子徐静，谢谢你的爱。由于译者经验和水平有限，译文难免有不妥之处，敬请读者指正并与我交流：develop_game@yahoo.com.cn。

徐明亮

前　言

本书将告诉你如何使用符合C语言的ANSI/ISO标准的库函数。因为已经有很多书出色地讲解了C语言本身，所以本书只专注于“库”这个话题。本书还会告诉你C标准库是如何实现的。本书提供了大约9 000行测试过的可实际工作的代码。我相信，看了C标准库的实现细节后你能更好地理解如何使用它。

库函数的实现代码尽可能地使用标准C，这样做有3个设计目的：首先，它使代码具有可读性和示范性；其次，它使代码在各种计算机体系结构间具有高度可移植性；最后，它能使编写的代码兼顾正确性、性能和规模各方面。

教你如何编写C程序并不是本书的目的。本书假定你能读懂简单的C程序，对于那些稍有难度的代码，我会向你解释其中的难点和技巧。

C标准库

C标准库是非常强大的，它在多种不同的环境下提供了相当多的功能：它允许用户和实现者使用明确定义的名字空间；它对其所提供的数学函数的健壮性和精确性有非常严格的要求；它率先对适应不同文化习惯的代码提供支持，包括那些拥有很大字符集的文化习惯。

为了能有效利用标准库所提供的强大功能，用户应该了解其实现上的很多隐晦细节。库的实现者必须向用户提供这些细节，以使他们更好地使用标准库。C标准中并没有把这些隐晦的实现细节都很好地描述清楚，因为制定标准的主要目的并不是给库的实现者提供指导。与ANSI C标准一起发布的Rationale也没有对这些细节作出很好的解释。Rationale要服务的对象范围很广，而关注这些细节的标准实现者只是众多服务对象的一部分。

在C的传统实现中并不能找到上面提到的新特性。现在的实现已经可以支持国际化开发中的区域设置（locale）概念。每个区域设置都对应于专属的某个国家、某种语言或者某个职业的特定习惯，一个C程序可以通过修改和查询区域设置来动态地适应多种文化。现在的实现也能支持很大的字符集，如字符数量众多的汉字。C程序能把它们作为多字节字符（multibyte character）或者宽字节字符（wide character）处理。它也能在这两种形式之间转换。在迅速加剧的市场竞争中，这就使得程序的编写更加简单和标准。

因为以前对这些新特性几乎不存在相应的编程艺术，所以即使是最有经验的C程序员，在使用区域设置、多字节字符和宽字节字符的时候也需要一些指导。所以，这些主题在这里给予

了特殊的关注。

细节

本书向用户和实现者解释了库的设计用意和可能用法。通过提供 C 标准库中所有库函数的实际实现，本书用例子告诉你怎样处理它的细节。在那些没有明确是最好实现方法的地方，它还讨论了可供选择和折中的办法。

一个涉及细节的例子是函数 getchar。头文件 <stdio.h> 原则上可以用下面的宏来屏蔽函数的声明：

```
#define getchar() fgetc(stdin)  /* NOT WISE! */
```

然而，它却不应该这样做，一个合理（即使没有用）的 C 程序是：

```
#include <stdio.h>
#undef fgetc

int main(void) {
    int fgetc = getchar();  /* PRODUCES A MYSTERIOUS ERROR */

    return (0);
    }
```

当然，这个例子有点极端，但它却阐述了即使是一个很好的程序员也可能犯的错误。用户有权要求尽可能少地出现这类奇怪的错误，所以设计者就有义务避免出现这些奇怪的错误。

我最终确定的 getchar 宏的形式是：

```
#define getchar () (_Files[0]->_Next < _Files[0]->_Rend \
    ? *_Files[0]->_Next++ : (getchar) ())
```

它和上面第一次给出的那个显而易见（而且更具可读性）的形式大不相同。第 12 章会解释其中的原因。

库的设计

本书还有一个目的，那就是从一般角度教授程序员怎样设计和实现库。从本质上讲，程序设计语言提供的库是一个混合的“袋子”。库的实现者要具备非常广泛的技巧才能处理这个“袋子”中各种各样的内容。仅仅是一个有能力的数值分析员，或者能熟练高效地操作字符串，或者懂得很多操作系统接口方面的知识，都是不够的。编写库不仅需要具备以上所有能力，还需要掌握更多的知识。

已经有很多好书告诉你怎样编写数学函数，也有很多书专门介绍某种特定用途的库。这些都是告诉你怎样使用现有的库。有些书甚至证明某个库的各种设计方案抉择的正确性。很少有书致力于告诉读者全面地构建一个库要求具备的技能。

可复用性

很多书都介绍了设计和实现软件的一般性原则。这些书中提到的方法有结构化分析、结构化设计、面向对象设计和结构化编程等。这些书中的大部分例子只考虑了为某个应用编写的程序。然而，这些原则和方法也同样适用于可复用的库的实现。

但可复用性的目标进一步提高了要求。从结构化设计的角度来看，如果一个库函数没有很高的内聚性，它就不太可能有新的用途。同样，如果它不具备低耦合性，它就会更难使用。简单地说，库函数一定要隐藏它的实现细节并且提供完整的功能。否则，从面向对象的角度看，它们不能实现可复用的数据抽象。

所以本书的最终目的是讨论构建库所特有的设计和实现问题。C 标准库的设计是固定的。然而，它从很多方面来看都是一个好的设计，值得讨论。C 标准库的实现也可以有很多选择。任何一个选择都要遵守一定的原则，例如正确性和可维护性等。其他的一些选择还要遵循某个项目特定的优先级，例如高性能、可移植性或者小规模等。这些选择和原则也值得讨论。

本书的结构

本书的结构基本对应 C 标准库本身。标准库中 15 个头文件声明或定义了库中所有的名字。本书中每一章讲述一个头文件。大部分头文件都有比较紧凑的内容，书中对此也作了紧凑的讨论。然而，也有一两个很笼统，它们相应的章节自然也会涉及更多的内容。

在本书的每一章中，我都摘录了 ISO C 标准的相关部分。（除了格式细节，ISO 和 ANSI 的 C 标准是完全相同的。）这些摘录补充说明了库的每一部分通常是怎样使用的。它们也使本书成为一个更完整的参考手册（当然本书比单独阅读 C 标准要容易理解）。本书也给出了实现那些部分和测试这些实现所需的代码。

每一章最后都附有参考文献和配套习题。假如本书作为大学的教材使用，这些习题可以当成作业。很多习题都只是简单的代码改写，它们可以把某个知识点讲得更清楚或者能够说明在实现上可以做一些合理的修改。那些更具有挑战性的题目都做了标记，可以将它们作为那些长期项目的基础，而自学者则可以把这些习题当作深入思考的入口点。

代码

本书中出现的代码已经在 Borland、GNU 项目和 VAX ULTRIX 的 C 编译器上进行了测试。它通过了广泛使用的 Plum Hall Validation Suite 的库函数的测试，也通过了那些专门为 C 实现设计的、发现 C 实现的缺陷的公共域程序。虽然我尽最大努力使错误减到最少，但仍然不能保证代码完全正确。

同时也请注意本书的代码是受版权保护的。它没有放置在公共域中，也不是共享软件。和那些自由软件基金会（GNU 项目）发布的代码不同，它不受“copyleft”协议的保护。我保留

本书所有的权利。

合理使用

你可以把书中的代码转变为电子版的形式，供个人使用。你也可以从堪萨斯州劳伦斯市的C用户组那里购买机器可识别的代码。无论哪种情况，你对代码的使用都要遵守版权法关于“合理使用”的规定。合理使用不允许你以任何形式发布这些代码的副本，不论是打印件还是电子版，也不论是免费的还是收费的。

除了以上谈到的，我允许合理使用范围之外的一种重要用途。你可以编译库中的部分并且把生成的二进制目标模块和你自己的代码连接生成可执行文件。无限发布这样的可执行文件是允许的。对这样的副本我不要求任何权利。但是，我强烈要求你声明库（在你的文件中）的存在，不论你用了多少，也不论是修改过的还是没有修改的。请在可执行文件的某个地方把下面的文字包含进去：“本作品的部分代码源自 *The Standard C Library*，copyright (c) 1992 by P. J. Plauger, Prentice-Hall 出版社出版，这些代码已获授权使用。”随可执行文件一同发布的文档中也要在适当的地方显著地标注这些内容。如果你遗漏了其中的任何内容，将被视为侵权。

许可

你也可以被授权做更多的事。你可以以二进制目标模块的形式发布整个库，甚至可以发布本书中的源文件的副本，不论是否修改过。简单地说，你可以把整个库合并到一个让人们使用它来编写可执行程序的产品中。但是，所有的这些都需要许可证。你可以购买许可证。如果你想购买许可证而且想不断地获得库的支持，请和夏威夷州卡姆艾拉市的 Plum Hall 公司联系。

虽然上面的几段有很浓的商业色彩，但我的主要目的不是推销一个商业产品。我非常信任C标准，也已经很努力地参与了它的制定。我花了很大的精力制定C标准库规范。我想证明我们创建了一个良好的语言标准。编写这个实现和这本书就是为了说明这个简单但很重要的事实。

致谢

马萨诸塞州韦克菲尔德市的 Compass 公司在本书还没完成的时候就很信任我。他们是我的库代码的第一批使用者，他们帮助我测试、调试，并且在 Intel 860 编译器上使用的过程中对这些库代码进行了改进。特别是 Ian Wells，愿意忍受我的延迟和修改，体现了他良好的专业精神。Don Anderson 先生为了使这个库的结构更合理，牺牲了很多休息时间给我发电子邮件。我衷心地感谢 Compass 公司的每个同事的真诚和耐心。

Prentice-Hall 的出版人 Paul Becker 对本书也很有信心。他的温和与不懈的激励是本书完成的一个保证。他请了一些匿名的审稿人帮助我，使我的观点更加明确，同时也减少了那些绝对化的语句。Paul 的职业精神使我明白了 Prentice-Hall 能在技术出版领域辉煌这么多年的原因。

写作过程中我搬到了澳大利亚，这个项目面临着重重困难。我的好朋友，澳大利亚 White-

smiths 公司的商业伙伴 John O'Brien 经常帮助我。他是一个"把荆棘变成玫瑰"的好手。他对我的帮助很大，不是友谊这个词能概括的。

感谢 Prentice Hall 的发行经理 Andrew Binnie 为我提供了完成这本书所用的激光打印机，他乐于在很多方面帮助我。感谢新南威尔士大学计算机科学系给我提供时间和场所，尽管他们也需要这些来完成他们自己的计划。

Tom Plum 一直告诫我们要深刻理解 C 的基础。我也多次从和他对本书中涉及的话题进行的讨论中受益匪浅。Dave Prosser 也与我分享了他对 C 的工作方式的深刻见解。作为 ANSI 和 ISO C 标准的编辑，Dave 提供了本书中大量电子版的文本资料。日本东京的高级数据控制公司（率先对 C 提供日文中的汉字的支持）的两个主要负责人 Hiroshi Fukutomi 和 Takashi Kawahara 在我了解日本程序员的技术需求的工作中，给了我很多帮助。

这里的大部分材料原来都在 *The C Users Journal* 上发表过。Robert Ward 是一个很容易相处的出版人，我很感谢他能让我重新使用这些材料。也同样感谢 Jim Brodie 允许我使用我们合著的 *Standard C* 一书中的内容。

阅读技术类书稿是一项很枯燥的任务。John O'Brien 和 Tom Plum 审阅了本书的部分内容并且反馈了很有价值的信息。发现第一次印刷版本中（只是一小部分）错误的人有：Nelson H. F. Beebe、Peter Chubb、Stephen D. Clamage、Steven Pemberton、Tom Plum 和 Ian Lance Taylor。

最后，我要感谢我的家人对我的支持。我的儿子 Geoffrey 在本书的排版和页面设计方面做了很多工作。我的妻子 Tana 在这么长的时间里给我提供了巨大的精神支持和后勤保障。他们的参与使我的写作过程充满乐趣。

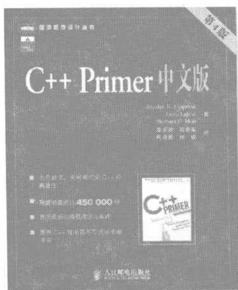
P. J. Plauger
邦迪，新南威尔士

TURING 图灵教育

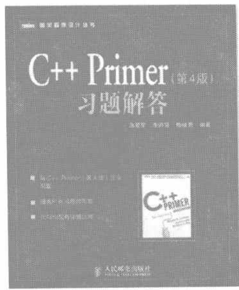
精选大师名作　打造专业品牌

人民邮电出版社 POSTS & TELECOM PRESS

人民邮电出版社图灵公司致力于出版国内外优秀科技图书，出版方向包括计算机、数学和电子电气等。几年来，图灵公司本着实事求是、精益求精的精神，出版了大量脍炙人口的经典著作，已成为高端技术图书出版领域的一面旗帜。

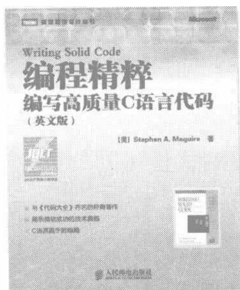


- 久负盛名的 C++ 经典教程
- 被 china-pub 会员评为“2007 年我最喜爱的十大技术图书”之一
- 被《程序员》评选为“2006 年最受读者喜爱的十大 IT 图书”之一

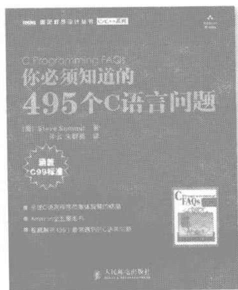


- 为 C++ Primer（第 4 版）的配套书籍

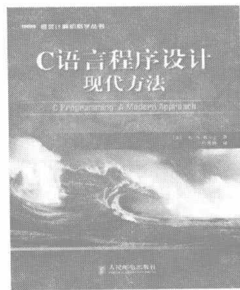




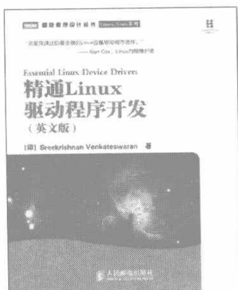
- 与《代码大全》齐名的经典著作
- 揭示微软成功的技术奥秘
- C 语言高手的秘籍



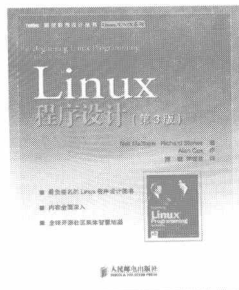
- 全球 C 语言程序员集体智慧的结晶
- Amazon 全五星图书
- 权威解答 495 个最常遇到的 C 语言问题



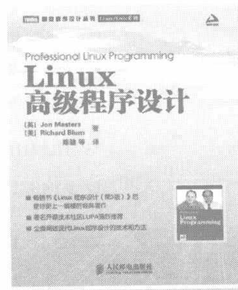
- C 语言领域的经典之作
- 近 10 年来最好的一部 C 语言著作
- 已被全球 200 多所学校采用为教材



- 目前最全面深入的 Linux 设备驱动程序著作
- 世界级 Linux 技术大师力作
- 大量技术内幕首次公开



- 最负盛名的 Linux 程序设计图书
- 内容全面深入
- 全球开源社区集体智慧结晶



- Linux 程序设计领域的一部力作
- 著名开源技术社区 LUPA 强烈推荐
- 全面阐述现代 Linux 程序设计的技术和方法

北京图灵文化发展有限公司　　电话：010-51095186　　传真：010-52086950　　网址：http://www.turingbook.com
地　址：北京市朝阳区北苑路13号院1号楼领地OFFICE C座603室　　邮编：100107　　E-mail：contact@turingbook.com

目　　录

第 0 章 简 介

0.1 背景知识

库（library）是一个可以在许多程序中复用的程序组件的集合。大部分程序设计语言都包含某种形式的库，C 语言也不例外。C 语言从一开始就包含许多有用的函数。这些函数可以帮助你进行字符分类、字符串操作、读输入和写输出等——这里只举出了一些类别的服务。

一些定义

在程序中使用函数之前，必须先对它进行声明。最简单的方法是把一个头文件（header）包含到程序中，该头文件声明了某种类别的所有库函数。头文件也可以定义任何相关的类型定义（type definition）和宏（macro）。头文件与函数本身都是库的组成部分。在大多数情况下，与构成程序的源代码一样，头文件也是文本文件（text file）。

你可以在 C 源文件中使用预处理指令 #include 将头文件变为翻译单元（translation unit）的一部分。例如，头文件 <stdio.h> 中声明了一些完成输入 / 输出操作的函数。使用函数 printf 打印一条简单消息的程序构成了下面的 C 源文件：

```
/* a simple test program */
#include <stdio.h>

int main (void)
    {    /* say hello */
    printf("Hello\n");
    return(0);
    }
```

翻译器（translator）把每一个翻译单元转换成一个目标模块，以适用于某一特定的计算机体系结构（或机器）。链接器（linker）把构成整个程序的所有目标模块组合起来，它也将用到的、来自 C 语言库的任何目标模块包含进去。编译器（compiler）是最常用的一种翻译器，它将源文件生成一个可执行文件。至少在理想情况下，一个可执行文件中只有包含着程序实际用到的那些函数的库所生成的目标模块。那样，你的程序就不会随着 C 语言库的扩充而增大。（解释器是另一种翻译器，它可能把整个 C 语言库都包含进来，以解释你的程序。）

创建库

你可以构建自己的库。典型的C编译器有一个库管理器（librarian），它可以将你所指定的目标模块汇集成一个库。链接器知道如何从任何库中只选取程序所用到的目标模块。C语言库并没有什么特殊的。

可以用C语言编写一个库的部分或者全部。编写用来生成库的目标模块的翻译单元也没什么特别的：

- 库的目标模块不应该包含具有外部连接属性的main函数定义。程序员不太可能复用一段总是在程序启动处接管控制的代码。
- 目标模块应该仅包括那些容易声明和使用的函数。提供一个头文件，该文件声明了这些函数且定义了相关的类型和宏。
- 最重要的是，库的目标模块应该在各种上下文环境中都可用。编写高度可复用代码是一种只能通过实践和学习成功的库培养出来的技能。

读完本书后，你将能用C语言轻松地设计、构建和编写专用的库。

用C编写的C语言库

C语言库本身就是用C语言编写的，而以前其他程序设计语言的库实现就往往不是这样。早期语言的库是用汇编语言（assembly language）实现的，而不同的计算机体系结构有不同的汇编语言。为了把这种库移植到另一种计算机体系结构下，必须将全部的库实现重写一遍。使用C语言可以编写出具有高度可移植性的健壮且高效的代码。你只需使用不同的C翻译器简单地翻译一下C代码，就可以把它移植到另一种机器上。

例如，下面是声明在<string.h>头文件中的strlen函数的实现，该函数返回一个以空字符（null）结束的字符串的长度，其指针参数指向字符串的第一个元素：

```
/* strlen function */
#include <string.h>

size_t (strlen) (const char *s)
    {   /* find length of s[] */
    const char *sc;

    for (sc = s; *sc != '\0'; ++sc)
        ;
    return (sc - s) ;
    }
```

strlen是一个很容易编写的小函数，也很容易在许多小地方出错。strlen的应用很广。你也许想为一个指定的计算机体系结构提供一个特定的版本，但实际上没有必要。这个版本就是正确的、可移植的，而且效率很高。

很多其他的现代语言都不能用来编写其自身语言库中的重要部分。例如，你不能用可移植的Pascal实现Pascal的库函数writeln。但是，你能用可移植的C语言实现C的库函数printf。这种对比有点不公平，因为C的类型检查没有Pascal那么强。然而，这一点非常重要——C语言库从最开始

就几乎完全是用C语言来表达的。

不可移植的代码

有些库函数的代码不能用*可移植的*C语言编写。用C语言编写的代码也许可以在很多种计算机体系结构下工作，但不是全部。在这种情况下，最重要的是要清楚地将那些有可能需要修改的不可移植部分记入。你也应该尽可能把不可移植的代码独立出来。即使是不可移植的C代码也比汇编语言更容易编写、调试和维护。应该只在某些不可避免的地方使用汇编语言，需要使用汇编的地方在C库中极少出现。

本书将告诉你如何使用目前的、标准化形式的C库。同时，也会告诉你如何用C语言编写C库，这有助于你理解库是如何工作的。本书也将从多方面举例说明如何用C设计和实现一个优秀的库。

0.2 C标准的内容

Dennis Ritchie于20世纪70年代初在AT&T贝尔实验室开发出了C语言的最初版本。开始它只是在DEC PDP-11计算机体系结构和特定的UNIX系统下实现的语言。然而，很快有人发现，它是以大多数现代计算机为模型而设计的。到20世纪70年代末，其他的几个C编译器作者已经在各类流行的计算机上实现了C编译器的不同版本，不论是微型机还是大型机。到20世纪80年代初，数以百计的C编译器版本已经被越来越多的程序员社区使用。这时，对C语言进行标准化的时候到了。

C标准

在美国，由美国国家标准化协会（ANSI）对程序设计语言进行标准化。X3J11（经ANSI授权的制定C标准的委员会的名字）始于1983年。现在的语言规范是ANSI标准X3.159-1989。[①]

与之类似，国际标准化组织（ISO）也开始在国际范围内制定C语言标准。ISO成立了JTC1/SC22/WG14技术委员会来对X3J11的工作进行审核和补充。目前，ISO采用了一个本质上与X3.159相同的标准ISO 9899:1990。这两种C标准只是在格式和各个部分的编号上略有不同，还有就是在一些细节地方使用不同的措辞，但这对语言标准的定义并没有本质的改变。

本书中广泛引用了ISO C标准。这样你就能准确地看到C标准所讲述的C标准库的方方面面。ISO是制定C标准的最高机构。如果你认为我的解释和C标准有出入，请以C标准为准。很有可能是我错了。

你经常会发现C标准难于阅读。其实，它这是有意用一种法律措词来阐释C标准。一个标准首先一定是准确和严密的，其次才考虑可读性。制定标准的文档也无意去充当用户的学习教程。X3J11也附随C标准提供了一个Rationale。如果你对X3J11所做的某些决定有些好奇，阅读一下Rationale也许会有所帮助。但是，需要强调的是，Rationale也不是关于标准C的学习教程。

① 最新的C语言标准是ISO/IEC 9899:1999（C99）。——编者注

下面有引自 ISO C 标准的两段话。第一段介绍了 C 标准的库部分。它提供了一些定义和几个重要的基本原则，这些原则从整体上影响了标准库的实现。

7. 库

7.1 介绍

7.1.1 名词定义

字符串

字符串是以第一个出现的空字符作为结束且包含该空字符的连续的字符序列。指向字符串的指针指向它的第一个（最低编址）字符。一个字符串的长度是空字符前面的字符的数目，它的值是它所含字符的值的相同顺序的序列。

字母

字母是指执行字符集中的可打印字符，它可以是 5.2.1 中列出的源字符集中的 52 个大小写字母中的任意一个。

小数点

小数点字符是对浮点数和字符序列进行转换的函数所使用的符号，用来指示这种字符序列的小数部分的起始位置[88]。它在文本和例子中用一个句点来表示，但可以被 setlocale 函数修改。

参见：字符处理（7.3）、setlocale 函数（7.4.1.1）。

7.1.2 标准头文件

标准头文件

每一个库函数都在一个头文件[89]中声明，头文件的内容可以通过一个 #include 预处理指令来使用。头文件声明了一组相关的函数，还包括使用的一些必需的类型和一些附加的宏。

标准头文件有：

```
<assert.h>  <locale.h>  <stddef.h>
<ctype.h>   <math.h>    <stdio.h>
<errno.h>   <setjmp.h>  <stdlib.h>
<float.h>   <signal.h>  <string.h>
<limits.h>  <stdarg.h>  <time.h>
```

如果有一个文件的名称和上面的尖括号中的任何一个字符序列相同，但不是作为语言实现的一部分提供的，而又放置在可供源文件包含的任何标准位置，那么这种行为是未定义的。

头文件可以以任何顺序被包含，每个头文件都可以在一个给定的范围内被包含多次，这和它被包含一次的效果相同，但包含 <assert.h> 的效果取决于 NDEBUG 的定义。如果要使用一个头文件，它应该被包含在任何外部声明或者定义的外面，而且应该在第一次引用它声明的任何函数或对象，或者它定义的任何类型和宏之前被包含。然而，如果一个标识符在多个头文件中定义或者声明，那么第二个和后来的头文件可以在对这个标识符的第一次引用之后被包含。程序在一个头文件被包含之前不应该有和当前定义的关键字重名的任何宏。

参见：诊断（7.2）。

7.1.3 保留的标识符

保留的标识符

每一个头文件都声明或者定义了在其相关联的标准条款①中列出的所有标识符，而且它还可选地声明或者定义相关联的库展望条款中列出的标识符和那些通常被保留下来作为文件范围标识符或其他用途的标识符。

- 所有以下划线加一个大写字母或者两条下划线开头的标识符都被保留作任何用途。
- 所有以下划线开头的标识符通常都在普通标识符和标记命名空间方面作为文件作用域标识符使用。

① 指 C 标准中说明相应头文件的条款，如 <assert.h> 的相关联条款是 C 标准中的 7.2 节。——编者注

- 后面的任何子句（包括库的展望）中列出的每个宏名都被保留作任何用途，如果它的任何相关头文件被包含。
- 后面的子句中（包括库的展望）所有具有外部连接的标识符都被保留为具有外部连接的标识符使用[90]。
- 每一个在后面的子句（包括库的展望）中列出的文件作用域标识符都被保留，作为一个相同命名空间中具有文件作用域的标识符来使用，只要它的任何一个相关头文件被包含了。

保留的标识符只有以上这些。如果一个程序声明或者定义了一个标识符，这个标识符和上下文环境中保留的标识符同名（不同于 7.1.7 中所允许的），那么这种行为是未定义的[91]。

脚注

88. 使用小数点字符的函数有 localeconv、fprintf、fscanf、printf、scanf、sprintf、sscanf、vfprintf、vprintf、vsprintf、atof 和 strtod。
89. 头文件不一定是一个源文件，头文件名字中由尖括号括住的字符序列也不一定是有效的源文件名字。
90. 具有外部连接的保留标识符包括 errno、setjmp 和 va_end。
91. 因为宏名只要存在，就会被代替，而和作用域和命名空间无关，所以，如果相关的头文件被包含了，和任何保留的标识符的名字相匹配的宏名一定不能定义。

第二段引用描述了 C 标准库里库函数的使用方式。

7.1.7 库函数的使用

库函数的使用

下面的每一个语句都成立，除非后面有另外明确的详细描述。如果一个函数参数有一个无效值（例如，函数的定义域外的一个值，或者程序的地址空间外的一个指针，或者一个空指针），那么这种行为未定义。如果一个函数参数被描述为一个数组，那么实际传递给函数的指针应该有一个值，这样所有的地址计算和对象访问（如果指针确实指向这个数组的第一个元素，那么它就是有效的）实际上都是有效的。在一个头文件中声明的任何一个函数可能又作为该头文件中的一个宏实现了。所以，即使它的头文件被包含了，它的库函数也不一定被显式声明了。一个函数的任何宏定义都可以通过用括号把函数的名字括住来局部地抑制它，因为这个名字后面没有跟着指示一个宏函数名展开的左括号。同理，即使一个库函数也作为宏被定义了[95]，获得该函数的地址还是允许的。用来移除任何宏定义的 #undef 预处理指令的使用也可以保证实际函数的引用。一个作为宏实现的库函数的任何调用都会一次展开为精确地计算它的每一个参数的代码，必要时可以完全用括号保护起来，所以，通常情况下，使用任何表达式作为参数都是安全的。同样，那些在下面子句中描述的类似于函数的宏可能会在一个表达式中调用，这个表达式可以在具有可兼容的返回值类型的函数被调用的任何地方[96]。所有展开为整型常量表达式的类似于对象的宏都适用于 #if 预处理指令。

如果不引用一个头文件中的任何类型定义就可以声明一个库函数，那么显式或者隐式地声明这个函数都是允许的，并且不用包含它的相关头文件就可以使用它。如果一个接收可变参数列表的函数没有被声明（显式地或者通过包含相关头文件），那么它的行为未定义。

例子

函数 atoi 可以通过以下几种方式使用：

- 通过相关头文件的使用（可能生成一个宏展开）。

```
#include <stdlib.h>
const char *str;
/*...*/
i = atoi(str);
```

- 通过相关头文件的使用（确实生成了一个真正的函数引用）。

```
#include <stdlib.h>
#undef atoi
const char *str;
/ *...* /
i = atoi(str);
```

或者

```
#include <stdlib.h>
const char *str;
/ *...* /
i = (atoi) (str);
```

- 通过显式声明。

```
extern int atoi(const char *);
const char *str;
/*...*/
i = atoi (str);
```

- 通过隐式声明。

```
const char *str;
/*...*/
i = atoi (str);
```

脚注

95. 那就是说，一个实现必须为每个库函数提供一个实际的函数，即使它已经为该函数提供了一个宏。
96. 因为外部标识符和某些以下划线开头的宏名是保留的，实现可能对这些名字提供特定的语义。例如，标识符 BUILTIN_abs 可以用来说明函数 abs 内联代码的产生。因此，正确的头文件能为一个其代码生成器可以接受这个标识符的编译器指定如下内容。

```
#define abs (x) _BUILTIN_abs(x)
```

通过这种方式，如果一个用户希望保证像 abs 这样的给定库函数是一个真实的函数，他可能会这样编写：

```
#undef abs
```

而不论实现的头文件是否提供了 abs 的一个宏实现或者内置实现。因而，前面被任意宏定义掩盖的函数原型也因此显露出来了。

引用 ISO 标准

注意，对引自 ISO C 标准中的每个语句我都做了明显的标记。它们的字体和正文相同，字号比正文小。我在引用的左边画了一条粗线（那条线左边的注释是我做的）。每个引用都包含了至少一个编号，以明确它在 C 标准中的位置。我收集了引用中要用到的所有脚注并把它们列在引用的最后。

我对 ISO C 标准摘录部分进行了排版，所使用的（机器可识别）文本与 C 标准的文本是相同的。当然，换行和换页不同。但是，需要提醒的是，我编辑了该文本，大量修改了其中的排版标记，因此可能引入了一些错误，而

后来校对的时候并没有发现这些错误。所以，还是那句话，C 的最终权威是你从 ISO 和 ANSI 那里获得的印刷版的 C 标准。

0.3 库的使用

C 标准对于用户应如何使用 C 标准库有很多说明，其中最重要的两点是：

- 怎样使用库的头文件；
- 怎样在程序中创建库名。

头文件的使用

C 标准库提供了 15 个标准头文件。任何预定义但没有在 C 语言中定义的符号名都在一个或多个标准头文件中定义了。这些头文件具有以下几个特性。

- *具有幂等性*。可以多次包含相同的标准头文件，但效果与只包含一次相同。
- *相互独立*。任何标准头文件的正常工作都不需要以包含其他标准头文件为前提。也没有任何标准头文件包含了其他标准头文件。
- *和文件级别的声明等同*。必须先把某标准头文件包含到你的程序中，然后才能使用该头文件已定义或声明的东西。不能在声明中包含标准头文件。并且，也不能在包含标准头文件之前用宏定义去代替关键字。

在 C 程序员中所达成的一个约定是：C 源文件的开头部分要包含所有要用到的头文件。在 `#include` 指令之前只能有一句注释语句。你可以按任何顺序书写这些头文件——我喜欢把它们按字母表的顺序排列。不要理会 C 标准有关函数声明的其他方式。

你的程序可能会用到你自己的头文件。不要用任何标准头文件的名字为你的头文件命名。也许你会在某个系统上侥幸成功，但换一个系统你就会遭遇失败。一个广泛使用的惯例是，按以下形式对 C 源文件名和头文件名命名：

- 以小写字母开头；
- 接着使用 1 ～ 7 个小写字母和数字；
- 以 `.c` 作为 C 源文件名的后缀，`.h` 作为头文件名的后缀。

例如 `i80386.h`、`matrix.c` 和 `plot.h`。这些形式的文件名适用于大多数的 C 编译器。除第一个字母外，你甚至可以通过使用至多 5 个另外的小写字母和数字来获得更好的可移植性，这是 C 标准所建议的。然而，我发现，这些较长的文件名的可移植性（和隐藏性）已经很好了。

你写的头文件可能会用到标准头文件中的定义或者声明。如果是这样的话，你最好把标准头文件包含在你所写的头文件的开头。这就消除了 C 程序源文件中包含头文件的顺序问题。不要为在一个翻译单元中多次包含同一标准头文件而担心，这正是可利用头文件幂等性的地方。

对你自己的头文件使用另一种形式的 #include 指令是个好习惯：用双引号包含头文件名，而不是用尖括号，只有引用标准头文件时才使用尖括号。例如，你可能在一个 C 源文件的开头写入以下内容：

```
#include <stdio.h>
#include "plot.h"
```

我的做法是首先列出标准头文件。但如果你按照前面的建议做，就无需这样做了。我遵循这种做法只是想把随意性减到最小。

命名空间

C 标准库有非常清晰的命名空间。库定义了 200 个外部名字。除此之外，它还保留了某种类别的名字供库的实现者使用。语言的使用者可以使用除此之外的所有名字。图 0-1 展示了 C 程序中存在的命名空间，该图出自 Mauger 和 Brodie 的 *Standard C* 一书，它告诉我们可以定义一套自由的命名空间。

- 每个块（函数中以大括号包含的部分）引入了两个新的命名空间。一个包括了所有作为类型定义、函数、数据对象和枚举常量声明的名字，另一个则包括了所有枚举、结构和联合的标记。
- 你定义的每一个结构或联合都引入了一个新的命名空间，其中包括了所有成员的名字。
- 你声明的每一个函数原型引入了一个新的命名空间，其中包括了所有的参数名。
- 你定义的每一个函数引入了一个新的命名空间，其中包括所有标号的名字。

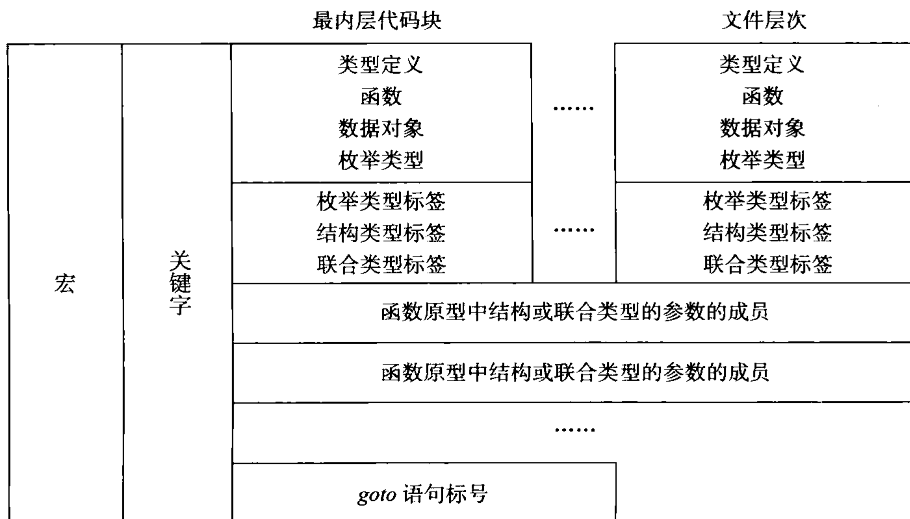


图 0-1
命名空间

在一个给定的命名空间中你只能以一种方式使用一个名字。如果翻译器识别出一个名字属于某个命名空间，它就看不到这个名字在另一个命名空间的其他用途。图 0-1 左边的命名空间块会掩盖住其右边的命名空间块。因此，宏可能会掩盖住关键字。这两者之间的任何一个都能掩盖其他类型的名字。(例如，你不能把数据对象的名字定义为 while。)

在实际使用中，你应该把所有的关键字和库名作为所有命名空间中的保留字。这会使你和代码的未来读者可能遇到的混淆减到最少。只有忘记了库中一个很少用到的名字时，才能依靠单独的命名空间来挽回。如果必须做一些冒险的事情，比如定义一个宏来代替一个关键字，那么一定要很小心并且清楚地记录下这些项目。写程序的时候一定不要使用以下某些类型的名字，它们是留给库的实现者使用的。

- 以下划线开头且具有外部连接属性的函数和数据对象的名字，例如 _abc 或者 _DEF。
- 以两个下划线或一个下划线加一个大写字母开头的宏名，例如 __abc 或者 _DEF。

记住，宏的名字可以掩盖其他任意命名空间的名字，第二类名字在所有的命名空间中都被有效地保留。

0.4　库的实现

本书后面出现的代码是在一些假设的基础上写出来的。如果想在一个指定的 C 实现上使用这些代码，必须确定这些假设在这些实现上是成立的。

假设

- *可以用一个同名的 C 源文件代替标准库头文件，例如 assert.h。*编译器可以把标准头文件的名字作为保留字，包含一个头文件可以很容易地打开构建在翻译器中的一系列定义。这样的编译器可能会带来一些问题。
- *可以代替不完整的标准头文件。*你可能只想测试这里给出的代码的一部分。即使你最终想把它们都试一下，也不会马上完成所有的工作。
- *可以用包括了这个函数的常规定义的 C 源文件替换预定义的函数。*编译器可以把库函数的外部名字作为保留字，调用一个库函数可以很容易地扩展为内联代码。这样的编译器可能会带来一些问题。
- *可以一点一点地替换一些预定义的函数。*编译器可以把很多库函数连接到一个目标模块里面。这种方法也适用于代替标准头文件。
- *C 源文件名应该至多有 8 个小写字母，后面是一个点和一个小写字母。*这是我在 0.3 节描述的形式。

- 外部名字可能把所有的字母和一种形式的字母相对应，也可能不是。这里给出的代码在两种情况下都能正确工作。

你的编译器不可能和所有这些假设不符；否则，只能通过使用一些技巧才能“合作”。大部分C的厂商都用C语言编写库，并且使用他们自己的翻译器。他们也需要这样做。

编程风格

本书的代码都遵循很多风格规范，其中大部分规范都适用于所有的项目，只有少数比较特殊。

- 库中的每一个可见函数都占据着单独的C源文件。文件名就是函数名（如果有必要的话，名字要变为8个字符）后面加上 .c 后缀。因此，函数 strlen 就在文件 strlen.c 中。在某些情况下，这会生成很小的文件。它简化了函数查找。附录B列出了库中定义的每一个可见函数名，同时也给出了定义该名字的文件所在的页码。
- 每个隐藏的名字都以一个下划线跟一个大写字母开头，如 _Getint。附录B也列出了所有有外部连接或在标准头文件中定义的隐藏的名字。
- 库中隐藏的函数和数据对象通常占据以 x 开头命名的C源文件，如 xgetint.c。这样的文件可以包含多个函数或数据对象，文件名源自它包含的某个函数或数据对象的名字。
- 代码排版也相当统一。我通常尽可能在嵌套里层的函数中声明数据对象。我用工整的缩进来说明控制结构的嵌套，我也在每个函数里面的左大括号（{）的后面写上一行注释。
- 代码不包括 register 声明，因为它们不容易安排并且会使代码混乱。此外，现代的编译器能比程序员更好地分配寄存器。
- 在一个库的可见函数的定义中，函数名被一对括号括住（参照0.1节 strlen 的定义）。所有这样的函数声明都会被相应的头文件中的宏定义所掩盖，所以这对括号阻止了翻译程序识别和展开宏。
- 本书中的C源文件都以一个带边框的图的形式给出，图的说明中有文件的名字。有些比较大的文件会出现在两个页面中，这样的图在每页的说明中都会提示你本页中出现的代码只是源文件的一部分。
- 每个图显示的C源代码都使用只占4列的水平制表符来缩进。显示的代码和实际的源文件有两点不同：代码右边的注释都经过调整而没有换行，每个C源文件的最后一行的结尾都用一个方框字符（□）做标记。

因而最后的代码有时候显得很密集。对于一般的编码项目，我会加一些空白使它至少增大 20%。为了使本书不会变得更厚，这些代码都是经过压缩的。

代码还包括很多应当合理地合并到一起的文件。就像上面提到的，把所有可见的函数放到单独的文件中会产生一些小得离谱的目标模块。我也额外引入了一些 C 源文件以保证所有的文件在两页的长度之内。但是，这并不是我压缩文件的唯一原因。开始我写的 C 源文件和它的自然长度相同，不管它有多大。我用的每一个编译器在翻译大文件的时候都至少有一个不能成功翻译。那些额外的模块有时可能称不上是好的设计，但它们在实践中确实提高了可读性和可移植性。

头文件的实现

本书中实现的 15 个源文件都是标准头文件。前面我列出了标准头文件的几个特性——幂等性、相互独立性和声明等价性。所有这些特性都会对你实现标准头文件有影响。

幂等性

幂等性很容易应付，对大多数的标准头文件可以使用*宏保护*。例如，你可以通过有条件地最多包含以下内容一次来保护 <stdio.h>：

```
#ifndef _STDIO_H
#define _STDIO_H
.../*BODY OF <stdio.h> */
#endif
```

当然，那个有趣的宏名 _STDIO_H 属于为实现者保留的一类名字。

对头文件 <assert.h>，你不能使用这种机制。它的行为受程序员可选择定义的宏名 NDEBUG 的控制。每次程序中包括这个头文件的时候，该头文件关闭或者打开宏 assert，这取决于翻译单元中的这个点是否定义了宏 NDEBUG。第 1 章中会进一步讨论这些内容。

相互独立性

有两个问题使得保持头文件之间的相互独立性有一点麻烦。一个是有些名字会在多个头文件中定义。一个程序应该可以包含两个定义了相同名字的头文件而不会造成错误。类型定义 size_t 就是一个例子。它是使用操作符 sizeof 所产生的类型（参照第 11 章)。你可以使用另一个宏保护来防止这种类型的多次定义。

```
#ifndef _SIZE_T
#define _SIZE_T
typedef unsigned int size_t;
#endif
```

宏 NULL 是另外一个例子。你可以在任何一个想引入指向数据对象的空指针（一个没有指派数据对象的指针值）的地方使用这个宏。定义这个宏的一种方式是：

```
#define NULL (void *) 0
```

在一个翻译单元中包含这样的宏定义的多个实例不会造成任何损失。标准C允许宏的*良性重定义*。具有相同宏名的两个定义必须具有相同的*记号序列*。它们只能是记号之间的空白不同（这里的空白指空格符和制表符）。没有必要阻止包含两个符合这种情况的定义。

然而，你必须在多个地方提供相同的定义，这是一个烦人的维护问题。这里有两种解决办法：

- 在多个地方编写相同的定义。定义改变时，就要找出这个定义的所有实例。
- 把定义放在一个独立的头文件中，给这个文件起一个不会和程序员创建的文件名冲突的名字。在每个用到该定义的头文件中包含这个头文件。

我选择第二种解决方法（大多数情况），因为它简化了使库适用于不同的编译器所做的工作。

3个输出函数 vfprintf、vprintf 和 vsprintf 出现了一个相似但又不同的问题。当你想打印一些或全部可变参数表的时候，可以调用这些函数，它们接受一个可变参数表。这3个函数都在头文件 <stdio.h> 中声明，每个函数都有一个 va_list 类型的参数。但是该类型并没有在 <stdio.h> 中定义，它只在头文件 <stdarg.h> 中定义。这时该怎么办呢？

同义字

如果细心一点的话，你会发现答案很简单。头文件 <stdio.h> 一定要包含一个 va_list 类型的同义字，*同义字*有一个为宏保留的名字。在一个标准头文件内部表达这3个函数中的每一个函数的原型，做这么多就够了。（当然，实现者就面临着复制多个头文件中的可见定义和同义字的问题。）

作为程序员，没有定义 va_list 类型就去*使用*这些函数是很困难的。（可以实现，但或许不是一个好的方式。）这就意味着在任何想使用这些函数的时候，你可能会想包含头文件 <stdarg.h>。所以，这仍然是程序员的工作。编译器没必要（而且禁止）在每次程序包含头文件 <stdio.h> 的时候都把头文件 <stdarg.h> 拉进来。

文件级别的头文件

标准头文件最后的这个特性是完全有利于实现者的。程序员必须在允许文件级别声明的地方包含一个标准头文件。这就意味着预处理指令 #include 不能出现在另一个声明内的任何地方。大多数标准头文件必须包含一个或多个*外部声明*。这些只在某些上下文环境中允许。没有这个限制性条件，很多标准头文件都不能作为普通的C源文件编写。

0.5 库的测试

测试是一个永恒的话题。只有最小的函数能测试它的所有情况，即使这样，也不能测试它与使用它的大程序的所有交互情况。你不得不测试所有可能的输入值，或者至少尝试通过代码的所有可能的路径。如果你的目标是最终证明一个函数没有任何缺陷，那么你多数情况下不可能达到这个目标。

测试所有路径

一个低一点的目标是写出测试可执行代码的每一部分的测试。这和测试代码每个可能通过的路径有很大的不同。然而，这已经足够在很高的水平上确保代码实质上是正确的。想写出这样的测试，你必须知道：

- 代码应该做什么（规格说明）；
- 它是怎样做的（代码本身）。

那么你必须设计能测试规格说明的每个细节的测试。（我故意不说清楚“细节”都包括什么。）原则上，那些测试应该可以访问代码的每一个“角落”。每一条代码都应该对实现规格说明的某一部分有所帮助。在实践中，当你第一次分析规格说明的时候，你总是要加一些你没有预料到的测试。

结果是一个复杂的代码段和你想测试的代码紧紧地联系在一起。测试程序可能和待测试的程序一样复杂，或者有过之而无不及。那会使将来需要维护的代码数量加倍。一块代码的修改会引起另一块代码的改动。可以使用一块代码来调试另一块代码。只有当两块代码协调运行时，你才能说测试完成了——至少这个时候完成了。所有这些投入很大地提高了代码的可靠性。

规格确认

另一种测试形式是*确认*。这里，你的目标是证明代码在多大程度上满足了它的规格说明，而可以忽略任何实现细节。厂商会知道用户很难看到的程序的实现细节。厂商既对测试代码的内部结构感兴趣，也对其外部特征感兴趣。然而，客户应该主要关心一个产品是否满足了它的规格说明，特别是当他们在比较两种或多种相互竞争的产品的时候。

性能测试

还有一种形式的测试是*性能测试*。对很多人来说，性能就是速度、完美和简洁。但是，其他的因素会同样或者更多地影响性能。例如，对存储器和磁盘的要求，无论暂时的还是持久的，或者可预见的最坏情况下的耗时。好的性能测试：

- 测定与代码各种使用方式有关的参数；
- 能被独立的用户执行；
- 有可重现的结果；
- 对“足够好”有合理的标准；

- 对“优于一般”和“卓越”有令人信服的标准。

有太多所谓的性能测试都违反了这些原则中的大部分或者全部。很多测试都是测试那些容易测试的部分，而不是测试值得测试的部分。

给定有限的时间和资金，那些明智的厂商尽可能多地加大在这些形式的测试方面的投入。在代码测试之前应该设计一个测试计划，还应该把全面的测试作为项目的一部分。理想情况下，应该让不同的程序员来写代码和测试代码。还应该从外部资源里获得独立于厂商的测试系统。必须建立代码修改后重新测试的制度。和提交的代码一样，供测试用的机器也应该被定时维护。

在开发代码方面我十分认可这样的专业素质。然而，在对那种理想情况说了那么多之后，只能到此打住。这里给出的代码已经广泛地经过了几种现有的程序和组件的确认，但是我并没有开发一个测试程序来测试可执行代码的每一个部分。本书给出了太多的代码，所以增加一整套合适的测试实在是太难实现了。

简单测试

相反，我写了很多简单的测试程序。每个测试程序都测试了C标准库中每个标准头文件提供的部分或者全部功能。你会发现这些测试程序主要定位于外部行为。这就意味着，从本质上，它们组成了一个简单的确认系统。然而，它们有时也会闯进内部结构测试的领域。一些实现错误太普遍、危害性太大以至于我不得不测试它们。这些测试程序很少涉及性能测试。

在刚才说过的正当测试的前提下，大多数情况下这些测试非常粗略和简单。不过，即使简单的测试也能达到一定的目的。只需用几行代码就可以确认一个函数是否满足了它的基本设计目标，这又确保了实现的健全性。当你做一些修改的时候（修改是不可避免的），重复这些测试来保证你的实现仍然健全。简单的测试也值得编写和维护。

最好的简单测试有几个共同的特征：

- 打印出一个标准的确认消息，退出时返回成功的状态来报告执行的正确性；
- 识别出其他所有不可避免的输出，以把代码的读者遇到的混乱减到最小；
- 提供有趣的独立于实现的信息，否则这些信息很难得到；
- 不输出其他的东西。

我已经养成了在每个头文件的名字前面加一个t来构建测试文件的名字。因此，tassert.c测试头文件<assert.h>。它保证宏assert做它应该

做的工作。当一个诊断失败时它会告诉你库的输出。当结束对 <assert.h> 的测试时它会输出 SUCCESS testing <assert.h> 的提示信息。

就像在 tstdio1.c 和 tstdio2.c 中的一样，有些比较大的头文件要求有两个或者更多的测试程序。注意，每个这样的文件都定义了它自己的 main 函数。你可以把每一个文件和 C 标准库链接在一起，生成一个独立的测试程序。不要把这些文件中的任何一个添加到 C 标准库中。即使有些预定义的名字以字母 t 开头，我仍然选择它作为文件的第一个字母，因为它有助于记忆，并且这些文件名和库中所有固有的名字都不冲突。

0.6 参考文献

ANSI Standard X3.159-1989(New York: American National Standards Institute,1989). 这是最早的 C 标准，由 ANSI 授权的 X3J11 委员会开发。随 C 标准发行的 Rationale 解释了很多深入探究的决议。

ISO/IEC Standard 9899:1990(Geneva: International Standards Organization, 1990). 除了格式细节和各部分的编号不同，ISO C 标准和 ANSI C 标准都相同。本书中引用的是 ISO C 标准。

B.W. Kernighan and Dennis M. Ritchie, *The C Programming Language, Second Edition*(Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1989). 这本书的第 1 版作为 C 语言的事实标准流行了很多年。它也为 C 提供了一个非常好的全面的指南。第 2 版加入了 ANSI C 标准的内容，也是一本很好的指南。

P. J. Plauger and Jim Brodie, *Standard C* (Redmond, Wa.: Microsoft Press, 1989). 这本书为整个 C 标准提供了一个完整但又简洁的参考，它覆盖了语言和库两方面的内容。

Thomas Plum, *C Programming Guidelines*（Cardiff, N.J.: Plum Hall, Inc.,1989）. 这是指导 C 编程风格的极好的一本书。它在 194 ～ 199 页还对一阶正确性测试进行了讨论。

0.7 习题

0.1　下面哪一项最能说明为什么库要包含函数？

- ❐ 函数使用广泛。
- ❐ 通过生成内联代码函数性能可以得到很大的提高。
- ❐ 函数很容易编写且写的方式有多种。
- ❐ 函数很难正确编写。
- ❐ 写函数引入了一些有趣的挑战。

- ❐ 在过去的应用中函数很有用。
- ❐ 函数能提供很多相关性很小的服务。

0.2　编写一个包含下面这行代码的（正确的）程序：

```
x: ((struct  x *)x)->x = x(5);
```

描述 x 的 5 个不同的用途。你能举一个在程序中直接用到其中任意两个用途的例子吗？

0.3　考虑下面的序列：

```
double a[] = {1.0, 2.0};
double *p = a;
double sqr(x) {return (x*x);}
#define sqr(x) x*x
```

下面每个表达式的结果是什么？

```
sqr(3.0)
sqr(3)
sqr(3+3)
!sqr(3)
sqr(*p++)
(sqr)(3+3)
```

0.4　上述哪个表达式和函数调用的过程不同？

0.5　上述哪个表达式可以通过改变宏的定义纠正（错误），哪个不能？

0.6　如果标准头文件可以互相包含，你必须采用什么样的风格来避免出现问题？

0.7　[**难**] 如果标准头文件可以任意定义名字，程序员应该怎样做才能保证一个很大的程序从另一个实现中移过来时仍能正确地运行？

0.8　[**很难**] 描述一种实现方法：当包含标准头文件时，关键字被宏掩盖而不出错。

0.9　[**很难**] 描述一种实现方法，该方法允许标准头文件包含在函数定义内部或者一个源文件的任意位置。

第1章

<assert.h>

1.1 背景知识

头文件 <assert.h> 唯一的目的就是提供宏 assert 的定义。可以在程序中关键的地方使用这个宏来进行断言。如果一处断言被证明非真，我们希望程序在标准错误流输出一条适当的提示信息，并且使执行异常终止。（第 12 章会描述怎样写到流中。）因此，可以这样编写：

```
#include <assert.h>
    ...
    assert(0 <= idx && idx < sizeof a / sizeof a[0]);
    /* a[idx] is now safe */
```

如果遵守断言来编程，代码都会变得更简单。它不需要检查索引 idx 是不是在有效的范围内，因为断言会处理它们。如果调试程序的时候一个“不可能”的情况发生了，马上就会出现诊断信息。因此程序不会磕磕碰碰地运行，在以后也不会产生其他的问题。

请注意，这不是编写产品代码的最好方式。实战中程序异常终止绝非明智之举。无论相伴而产生的提示信息对程序员多么有用，它对用户都是天书。某种形式的错误恢复才应该是首选的方案。任何诊断信息应该都能被用户理解。

我们所需要的是这样一种方式，断言只在调试程序的时候起作用。这样从一开始就可以记录下需要的断言，让它们帮助尽早发现那些最糟的逻辑错误。稍后，可以加些代码来从程序执行时发生的错误中恢复。我们希望这些断言作为文档保留下来，同时又不希望它们产生代码。

宏 NDEBUG

<assert.h> 正好提供了这种行为。我们可以通过在程序的某些地方定义宏 NDEBUG 来改变 assert 的展开方式。如果程序中某个包含 <assert.h> 的地方没有定义 NDEBUG，该头文件就会将宏 assert 定义为活动形式，它就可以展开为一个表达式，测试断言并在断言为假的时候输出一条错误信息，然后程序终止。反之，如果定义了 NDEBUG，头文件就会把这个宏定义为不执行任何操作的静止形式。

1.2 C 标准的内容

<assert.h>

7.2 诊断 <assert.h>

头文件 <assert.h> 定义了宏 assert，还引用了另一个宏：

```
NDEBUG
```

后者不是 <assert.h> 定义的。如果 NDEBUG 在包含了 <assert.h> 的源文件中某处被定义为宏名，那么宏 assert 就被直接定义为：

```
#define assert(ignore) ((void) 0)
```

宏 assert 应该作为一个宏而不是一个实际的函数来实现。如果为了访问一个实际的函数而禁止了宏定义，那么这种行为是未定义的。

7.2.1 程序诊断

7.2.1.1 宏 assert

assert

概述

```
#include <assert.h>
void assert(int expression);
```

说明

宏 assert 向程序中加入诊断。当它执行的时候，如果表达式为假（就是说和 0 相等），assert 宏就按照实现定义的格式向标准错误文件写入关于这个特定调用的失败信息（包括参数的文本、源文件的名字和源文本行数——后面两者分别是预定义的宏 __FILE__ 和 __LINE__ 的值）[97]，然后它调用 abort 函数。

返回值

宏 assert 没有任何返回值。

参见：abort 函数（7.10.4.1）。

脚注

97. 写入的信息可能具有以下形式：

Assertion failed: *expression*, file *xyz*, line *nnn*

1.3 <assert.h> 的使用

在本章的开头有一个使用宏 assert 的例子。无论 assert 是活动的还是静止的，它的行为本质上都像是接受一个 *int* 参数而返回一个 *void* 类型结果的函数。该宏的参数表面上是一个整型表达式。如果表达式的值为零，宏就会写出一条信息并终止程序的执行。

谓词

实际编写的参数是一个谓词——一个真（非零）或者假（零）的表达式。在 *for*、*if* 和 *while* 语句中写入这些谓词来判断程序中的控制流。一个断言可以简写为：

```
if (! okay)
    abort();
```

函数 abort 在头文件 <stdlib.h> 中声明，当程序某些地方出错的时候，可以调用该函数来终止程序的执行。

断言可以帮助记录代码所作的假设，在调试代码的时候，它们也提供了那些假设的入口。然而，前面我就强调过，一个产品程序不能这样突然间终止。无论断言在调试的时候多么方便，最终它们都是多余的。

宏 NDEBUG

怎样控制宏的展开方式不过是个风格问题。但是必须通过某种方式来控制宏 NDEBUG 是否定义。一种编程方式是修改源代码。如果你认为断言没有存在的必要，则只需在包含头文件之前加上下面的代码就可以了：

```
#define NDEBUG /* disable assertions */
#include <assert.h>
```

这段代码清楚地说明了断言从此不再起作用。而当退回来重新调试的时候，它唯一的缺陷就会显露出来。（我敢保证一定会出现这种情况。）此时必须重新编辑源文件来移除该宏定义。

make 文件

很多编译器都支持一种更灵活的方法。它们允许在任意 C 源文件之外定义一个或多个宏。可以在一个命令脚本或重建程序文件的 make 文件中指定这些定义。在那里定义 NDEBUG 和说明断言无效更好。在必须重新回到一个更原始的调试阶段的时候，它也是一个更容易复制和修改的文件。C 标准没有对这样的功能作任何要求，但实现者在设计 <assert.h> 的时候还是考虑到了这一点。

这个头文件有个额外的特性。就像在前面的章节提到的那样，所有其他的头文件都具有幂等性。包含它们中的任意一个两次或者更多次，与包括一次具有完全相同的效果。但对 <assert.h> 来说，每次包含它的时候它的行为都会变化。该头文件会改变 assert 的定义来适应 NDEBUG 当前的定义状态。

因而实际的效果是你可以在整个源文件中用不同的方式控制断言。例如，当断言在频繁执行的循环内部发生时，性能可能会急剧下降。或者在到达提示性的部分之前，一个更早的断言可能会终止程序。不管哪种情况，都需要在一个源文件中各个不同的地方打开或者关闭断言。

所以要打开断言，可以写：

```
#undef NDEBUG
#include <assert.h>
```

要关闭断言，可以写：

```
#define NDEBUG
#include <assert.h>
```

良性重定义

注意：即使宏 NDEBUG 已经被定义了，我们仍然可以安全地定义它。正如 0.4 节所说的，这是一个良性重定义。在标准 C 中添加良性重定义就是这个目的。在有宏保护和条件语句的情况下，就不用再防止宏的多次定义了。

1.4 <assert.h> 的实现

这个头文件需要的代码很少，但必须非常细心地完成。为了对 NDEBUG 做出正确回应，该头文件一定要有一个总体结构：

```
#undef assert    /* remove existing definition */
#ifdef NDEBUG
#define assert (test) ((void) 0)  /* passive form */
#else
#define assert (test) ...          /* active form */
#endif
```

良性取消定义

如果当前宏 assert 的定义不存在，那么开头的 #undef 预处理指令也没有任何副作用。总是可以 #undef 一个名字，不论它是不是被定义为一个宏。（可以把这作为良性取消定义考虑。）但是，如果定义可能改变的话，这条预处理指令还是很有必要的。

一个简单地编写宏的活动形式的方式是：

```
#define assert (test) if (!(test)) \
  fprintf (stderr, "Assertion failed: %s, file %s, line %i\n", \
      #test, __FILE__, __LINE__) /* UNACCEPTABLE! */
```

这种方式因为各种各样的原因而不能接受：

- 宏不能直接调用库的任何输出函数，例如 fprintf。它也不能引用宏 stderr。这些名字只能在头文件 <stdio.h> 中正确地声明或者定义。程序可能没有包含这个头文件，但 <assert.h> 一定不能包含。一个程序如果不包含某个头文件，就可以定义宏来对该头文件中的任意名字重新命名。这就要求这个宏必须调用一个具有隐藏名字的函数来进行实际的输出。
- 宏必须能扩展为一个 *void* 类型的表达式。例如程序能包含一个形如 (assert(0<x),x<y) 的表达式，这就不能使用 *if* 语句了。任何测试都要在一个表达式内部使用某个条件操作符。
- 宏应该可以扩展为有效并且紧凑的代码。否则，程序员就会尽量避免使用断言。而这个版本却总是调用一个传递了 5 个参数的函数。

图 1-1 显示了文件 assert.h。宏 assert 的这种实现方法履行了测试方针。通过这种方式，一个优化的翻译器大都能清除明显正确的断言的所有代码。这个宏以 xyz:nnn 表达式（使用 C 标准的符号）的形式把诊断信息添加到一个字符串参数中。字符串创建操作符 #x 对大多数信息编码，然后字符串拼接程序把这些片断合并到一起。由于使用了 file 和 line，这比 C 标准建议的方式更紧凑。

_STR
_VAL

一个令人讨厌的问题是内置宏 __LINE__ 没有扩展成字符串字面量，它变成了一个十进制常量。把它转换为适当的形式需要一个额外的处理层。那

要通过向头文件中添加两个隐藏的宏 _STR 和 _VAL 来实现。其中一个宏用它的十进制常量扩展来取代 __LINE__，另一个是把十进制常量转换为一个字符串字面量。忽略 _STR 和 _VAL 中的任何一个，就会得到字符串字面量 "__LINE__"，而不是你想要的结果。

图 1-1
assert.h

```
/* assert.h standard header */
#undef assert                         /* remove existing definition */

#ifdef NDEBUG
    #define assert(test)    ((void)0)
#else                                         /* NDEBUG not defined */
    void _Assert(char *);
        /* macros */
    #define _STR(x) _VAL(x)
    #define _VAL(x) #x
    #define assert(test)   ((test) ? (void) 0 \
       : _Assert (__FILE__":"_STR(__LINE__)"" #test))
  #endif
```

函数 _Assert

图 1-2 显示了文件 xassert.c。它定义了宏调用的隐藏库函数 _Assert。一个巧妙的 _Assert 函数能解析诊断信息，也能补充缺少的位数。这里给出的版本不能提供这些功能，因为那些精确的信息格式是实现定义的。

图 1-2
xassert.c

```
/*_Assert function */
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>

void _Assert(char *mesg)
    {                          /* print assertion message and abort */
    fputs(mesg, stderr);
    fputs(" -- assertion failed\n", stderr);
    abort();
    }
```

前向引用

函数 _Assert 使用了两个其他的库函数。它通过调用 <stdio.h> 中声明的函数 fputs 把字符串写到标准错误流，通过调用 <stdlib.h> 中声明的函数 abort 异常终止程序的执行。有关这些头文件的描述会在较后面的章节出现。如果你对 C 有一个大体的了解，看这些前向引用应该没什么问题。但是如果你需要进一步了解这部分的函数的功能，你必须跳过很多页。

一个好的入门指南会把这些提前用到的知识的难度减到最小。但是，C 标准库是高度相互联系的。几乎每一部分的实现都以其他部分为依据而且只能依据其他部分来描述，就像我对 fputs 和 abort 的做法一样。当我必须提前提及的时候，我用一般的说法来表达这些新内容。这会让新接触标准 C 的读者尽量少翻来翻去，但是恐怕不能完全做到。

1.5 <assert.h> 的测试

图 1-3 显示了文件 tassert.c。这个测试程序通过 4 种不同的方式测试宏 assert——以它的静止和活动形式，包括满足测试条件和不满足测试条件的情况。只有不满足测试条件的活动形式才应该终止。正确的程序执行应该显示类似下面的东西：

```
Sample assertion failure message --
TASSERT.C:43 val == 0 -- assertion failed
SUCCESS testing <assert.h>
```

并且能正常终止。然而，注意，程序输出文本到标准错误流和标准输出流。文本行在不同的实现中可以按不同的顺序出现。（参照第 12 章关于流的讨论。）

图 1-3 tassert.c

```
/* test assert macro */
#define NDEBUG
#include <assert.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>

        /* static data */
static int val = 0;

static void field_abort(int sig)
    {                                               /* handle SIGABRT */
    if (val == 1)
        {                                           /*expected result */
        puts("SUCCESS testing <assert. h>");
        exit(EXIT_SUCCESS);
        }
    else
        {                                       /* unexpected result */
        puts("FAILURE testing <assert. h>");
        exit(EXIT_FAILURE);
        }
    }

static void dummy()
    {                                   /* test dummy assert macro */
    int i = 0;

    assert(i == 0);
    assert(i == 1);
    }

#undef NDEBUG
#include <assert.h>

int main()
    {                         /* test both dummy and working forms */
    assert(signal(SIGABRT, &field_abort) != SIG_ERR);
    dummy();
    assert(val == 0);                           /* should not abort */
    ++val;
    fputs("Sample assertion failure message --\n", stderr);
    assert(val == 0);                           /* should abort */
    puts("FAILURE testing <assert. h>");
    return(EXIT_FAILURE);
    }
```
□

如果前面提到的 assert 的 3 次调用中的任何一个导致程序异常终止，或者程序正常退出并返回 EXIT_FAILURE（<stdlib.h> 中定义的一个非零值）的状态报告，测试就会失败。

tassert.c 是一个相当成熟的测试程序。它使用的函数中有两个和我们见过的函数差不多。程序通过调用 <stdio.h> 中声明的函数 puts 把串写到标准输出流，通过调用 <stdlib.h> 中声明的 abort 来正常终止程序。然而，程序的功能不止这些。它通过调用 <signal.h> 中的 signal，在 _Assert 调用 abort 之后重新获得控制权。它甚至使用宏 assert 来验证 signal 返回了正确的状态。想象一下使用要测试的机器来实现测试工具！那简直不是调试新代码的方法。

程序桩　事实上，这不是我调试代码的方式。我的第一个版本的 tassert.c 在宏 assert 第四次测试的时候就以失败告终。我承认走到那一步已经进行了几次尝试。fputs 和 signal 都适用于很多机器，我开始测试 <assert.h> 时并没有在所有这些机器上进行调试。为了一次又一次调试代码方便，我不得不引入*程序桩*（一个简单得多的版本）。调试的必要性和简单的可靠性测试的必要性有很大的不同。当其中的一个测试失败的时候，你必须要改变它（或者依靠一个交互的编译器的服务）来准确地找到出错的位置。这是我为了保持测试简洁所做的设计妥协之一。

1.6 参考文献

两本提倡用断言编程的好书是：

O.J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, *Structured Programming* (New York: Academic Press, 1972).

E.W. Dijkstra, *A Discipline of Programming* (Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1973).

尽管这两本书都出版了很长时间，但它们仍然很受关注。

1.7 习题

1.1　使用图 1-2（图中的格式与 C 标准中的格式完全一致）中 xassert.c 的版本编写 assert.h 的一个版本。

1.2　使用图 1-1（图中的格式与 C 标准中的格式完全一致）中 assert.h 的版本编写 xassert.c 的一个版本。

1.3　上面两个练习的方法有什么相关优点？

1.4　写出 assert.h 和 xassert.c 的一个版本，要求能打印出所有断言。你为什么想使用该版本？

1.5 [**难**] 写一个处理控制信号 SIGABRT 的程序，该信号输出以下提示信息：Continue(y/n)? 到标准错误流，并从标准输入流读入响应。如果响应是 yes（大小写都可以），处理程序应该使自己复位，并把控制权还给 abort 函数。（第 9 章描述信号，第 13 章描述 abort 函数。）

为什么希望使用这种功能？

1.6 [**难**] 写一个处理控制信号 SIGABRT 的程序，该信号在 main 函数的顶部执行一个 longjmp 到一个 setjmp。（第 8 章描述 longjmp 和 setjmp 函数。）

为什么希望使用这种功能？描述一种安全的规则，该规则为使用这种功能的程序初始化一个静态存储空间。

1.7 [**很难**] 一些 C 翻译器提供一个*源文件级的交互式调试器*。这样的调试器通常允许你在正在执行的程序中的各个地方设置条件断点。找一个这样的 C 翻译器，探索一下需要什么条件才能使 <assert.h> 在这样的调试器下工作。目标是（按照难度递增的顺序）：

- 不论什么时候断言出错，都要把控制权回交给调试器。遇到符合宏 assert 的令人讨厌的调用语句时，程序能够继续执行。
- 不让宏 assert 产生内联代码。相反地，它应该传递指令到源文件级的调试器。
- 无论宏 assert 是否出现，也不论它是活动的还是静止的，生成最优级别的代码。
- 实现一个改进的宏 assert，它可以接受任意复杂度的测试表达式。

为什么希望使用以上每一个功能？

第 2 章

<ctype.h>

2.1 背景知识

字符处理从 C 发展的早期开始就很重要了。那时很多人被 DEC（数字设备公司）PDP-11 丰富的字符操作指令集所吸引。当 Ken Thompson 把 UNIX 移植到 PDP-11/20 时，他给我们提供了一种优秀的工具来以统一的方式操作字符流。C 出现以后，当编写具有强大字符处理功能的程序时，C 自然就成了首选的语言。

这是一种全新的程序设计风格。C 程序趋向于短小且致力于单一功能的编写。在这之前的传统是编写提供很多服务的大工程。C 程序可以读入和写出人类易读取的字符流，之前是通过高度结构化的二进制文件进行程序间的通信。它们利用附带的回车控制（embedded carriage control）产生标页数的报告来向人们提供信息。

惯用法 早期在 UNIX 下用 C 编写的工具很快就形成了很多惯用法。我们经常将字符排序并分成不同的类别。为了识别一个字母，可以编写：

```
if ('A' <= c && c <= 'Z' || 'a' <= c && c <= 'z')
    ...
```

当执行字符集是 ASCII 码（ASCII 是“American Standard Code for Information Interchange（美国标准信息交换码）”的缩写。它是一种使用广泛的字符编码，但并不是所有的体系结构都使用它。因此这种惯用法就不适用于其他常用的字符集，例如 IBM 的 EBCDIC）时，执行上述代码就可以得到正确的结果。

为了判别一个数字，可以编写：

```
if ('0' <= c && c <= '9')
    ...
```

为了判别空白，可以编写：

```
if (c == '' || c =='\t'||c =='\n')
    ...
```

很快，我们的程序因充斥着类似这样的判断语句而变长，尤其在几乎全部是这种判断语句的情况下，程序变得很长。我们也可以用一些其他方式来

实现上述惯用法，但那样更不容易理解并且发生错误的几率更大。

字符类别

对于某些字符分类方法也可以有多种选择。比如空格就受困于它众所周知的多变性。你可以将垂直制表符和水平制表符及空格混用吗？如果使用换行符（实际上是 ASCII 换行），也应该包含回车符（UNIX 保留的用来输出一行写满后的换行）吗？那么怎么处理换页？要想让各种工具能在一起协调工作，就要使它们遵守共同的规则。

我们很自然想到的方法是，引入函数来代替这些判断语句。这使它们立即变得更具可读性、更一致、更容易适应执行字符集中的变化。于是，上面的惯用法就变成了：

```
if(isalpha(c))
    ...
if(isdigit(c))
    ...
if(isspace(c))
    ...
```

一旦有很多这样的函数出现，程序代码将会变得很长。它们很快走入了正在成长的 C 所支持的函数库。越来越多的程序员使用这些函数，而不再是重复构造他们自己的惯用法。这些字符分类函数非常有用，也特别优秀，人们甚至开始怀疑这是不是幻觉。

过去，它们确实很好。一个典型的文本处理程序对输入流中的每个字符会平均调用 3 次这样的函数。因为这样的函数太多了，所以调用它们经常决定着程序的执行时间。这就导致一些程序员尽量避免使用这些标准字符分类函数。进而，其他的一些程序员就开发了一些宏来代替这些函数。

宏带来的意外

C 程序员更喜欢宏。因为宏可以让程序员编写出和函数调用一样可读但效率更高的代码。不过，要警惕一些意外情况：

- 虽然宏可能比函数调用执行得快，但它展开后的代码可能会比函数调用的多几倍。如果你的程序在很多地方扩展宏，这个程序可能大得让你无法想象。
- 宏可能会展开为一个子表达式，但它不像函数调用那样捆绑得那么紧凑。这是不能接受的，但又是经常发生的。在宏定义中使用圆括号可以消除这种弊端。
- 宏展开后，某些参数可能执行了不止一次，或者根本就不执行。一个具有副作用的宏参数会导致意外的发生。虽然一些 C 程序员认为这样的意外可以接受，但现代编程习惯还是避免使用它们。只有两个 C 标准库函数 getc 和 putc（都在 <stdio.h> 中声明），使用时可能会产生这种不安全行为的宏。

转换表

所以，在那些日子里，最大的挑战就是编写一个宏的集合来代替这些字符分类函数。因为使用范围很广，所以它们展开后要很紧凑。当然，使用的时候也要相当安全。久而久之，这些宏就演化成了使用一个或多个转换表（translation table）的宏集合，每个宏都有以下形式：

```
#define _XXXMASK   0x...
#define isxxx(c)   (_Ctyptab[c] & _XXXMASK)
```

字符 c 编入以 _Ctyptab 命名的转换表索引中。每个表项的不同位以索引字符为特征。如果任何一个和掩码 _XXXMASK 相对应的位被设置了，那个字符就在要测试的类别中。对所有正确的参数，宏展开成一个紧凑的非零表达式。

这种方法的弊端是对某些错误的参数，宏会产生错误的代码。如果一个宏的参数不在它的定义域内，那么执行这个宏时，它就会访问转换表之外的存储空间。这种错误可能会一直隐藏而不被发现，也可能会终止程序的执行，返回一个模糊的信息，这都取决于具体的实现。

这些函数假设它们测试的值是函数 fgetc、fputc、getc、getchar、putc、putchar 和 ungetc（都在 <stdio.h> 中声明）中某一个的返回值。所有这些函数都返回一个转换为 *unsigned char* 类型的字符代码——一个很小的非负值，或者返回宏 EOF 的值，该宏在 <stdio.h> 中声明，是一个负值（通常是 –1）。

在 *char* 类型和 *signed char* 类型表示含义相同的计算机体系结构下，当测试那些比较生僻的字符代码时，就会发生一个常见的错误。函数调用 isprint(c) 看起来很安全。但是若 c 为 *char* 类型，而且符号位被置位，那么参数会是一个负值，该值基本上是在函数的定义域之外。

几乎没有程序员会编写 isprint((unsigned char)c)，虽然它更安全。当然，在确认参数值 EOF 不会出现的地方，也可以安全地使用强制类型转换。

区域设置

然而，转换表仍然是很多字符分类函数的当今实现的基础。它们会为实现者提供高效的宏，即使在多区域设置出现的地方也是如此。区域设置是一个很长的话题，我会在第 6 章中详细地讨论它们。

现在，我只是发现一个 C 程序总是在 "C" 区域设置中开始执行。调用函数 setlocale 可以改变区域设置。当区域设置改变的时候，<ctype.h> 中声明的函数的某些属性就可能改变它的行为。

<ctype.h> 中声明的函数对当今的 C 程序员来说仍然是很重要的。你会在对字符进行排序分类的任何地方使用这些函数。它们可以让你在各种各样的字符集中更有可能编写出既高效又正确的代码。

2.2 C 标准的内容

<ctype.h>

7.3 字符处理 <ctype.h>

头文件 <ctype.h> 声明了几个可以用于识别和转换字符的函数 [98]。

对于所有参数是 int 类型的情况，参数值可以表示为 unsigned char 类型，或者和宏 EOF 的值相等。如果参数为其他的值，则它的行为未定义。

这些函数的行为受当前区域设置的影响。当处于 "C" 之外的区域设置时，有些函数的某些方面是由实现定义的，下面都把这些函数列了出来。

打印字符（printing character）指的是由实现定义的字符集的一个元素，每一个打印字符在显示设备上都占据一个打印位置。控制字符（control character）是实现定义的字符集中不是打印字符的元素 [99]。

参见：EOF（7.9.1）、区域设置（7.4）。

7.3.1 字符判断函数

本子句下的函数当且仅当参数 c 的值和函数描述中的一致时才返回非零（真）。

isalnum

7.3.1.1 函数 isalnum

概述

```
#include <ctype.h>
int isalnum(int c);
```

说明

函数 isalnum 判别所有 isalpha 或者 isdigit 判别为真的字符。

isalpha

7.3.1.2 函数 isalpha

概述

```
#include <ctype.h>
int isalpha(int c);
```

说明

函数 isalpha 判别所有 isupper 或者 islower 判别为真的字符，或者那些实现定义的字符集中的 iscntrl、isdigit、ispunct 和 isspace 判别都不为真的字符。在 "C" 区域设置中，isalpha 只对 isupper 或者 islower 判别为真的字符返回真。

iscntrl

7.3.1.3 函数 iscntrl

概述

```
#include <ctype.h>
int iscntrl(int c);
```

说明

函数 iscntrl 判别所有的控制字符。

isdigit

7.3.1.4 函数 isdigit

概述

```
#include <ctype.h>
int isdigit(int c);
```

说明

函数 isdigit 判别所有的十进制数字字符（见 5.2.1 中的定义）。

isgraph

7.3.1.5 函数 isgraph

概述

```
#include <ctype.h>
int isgraph(int c);
```

说明

函数 isgraph 判别除空格（' '）之外的所有打印字符。

islower

7.3.1.6 函数 islower

概述

```
#include <ctype.h>
int islower(int c);
```

说明

函数 islower 判别所有的小写字母或者实现定义的字符集中 iscntrl、isdigit、ispunct 和 isspace 判别都不为真的字符。在 "C" 区域设置中，islower 只对定义为小写字母（见 5.2.1 中定义）的字符返回真。

isprint

7.3.1.7 函数 isprint

概述

```
#include <ctype.h>
int isprint(int c);
```

说明

函数 isprint 判别包括空格（' '）在内的所有打印字符。

ispunct

7.3.1.8 函数 ispunct

概述

```
#include <ctype.h>
int ispunct(int c);
```

说明

函数 ispunct 判别除空格（' '）和 isalnum 判别为真的字符之外的所有打印字符。

isspace

7.3.1.9 函数 isspace

概述

```
#include <ctype.h>
int isspace(int c);
```

说明

函数 isspace 判别所有标准的空白字符，或者由实现定义的字符集中 isalnum 判别为假的字符。标准空白字符有：空格（' '）、换页（'\f'）、换行（'\n'）、回车（'\r'）、水平制表符（'\t'）和垂直制表符（'\v'）。在 "C" 区域设置中，isspace 只对标准空白字符返回真。

isupper

7.3.1.10 函数 isupper

概述

```
#include <ctype.h>
int isupper(int c);
```

说明

函数 isupper 判别所有的大写字母或者实现定义的字符集中 iscntrl、isdigit、ispunct 和 isspace 判别都不为真的字符。在 "C" 区域设置中，islower 只对定义为大写字母（见 5.2.1 中定义）的字符返回真。

isxdigit

7.3.1.11 函数 isxdigit

概述

```
#include <ctype.h>
int isxdigit(int c);
```

说明

函数 isxdigit 判别所有的十六进制数字字符（见 6.1.3.2 中定义）。

7.3.2 字符大小写转换函数

tolower

7.3.2.1 函数 tolower

概述

```
#include <ctype.h>
int tolower(int c);
```

说明

函数 tolower 把一个大写字母转换为相应的小写字母。

返回值

如果参数是 isupper 判别为真的字符，并且有一个和它对应的 islower 判别为真的字符，那么函数 tolower 就返回这个对应的字符，否则，返回原来的参数值。

toupper

7.3.2.2 函数 toupper

概述

```
#include <ctype.h>
int toupper(int c);
```

说明

函数 toupper 把一个小写字母转换为相应的大写字母。

返回值

如果参数是 islower 判别为真的字符，并且有一个和它对应的 isupper 为真的字符，那么函数 toupper 就返回这个对应的字符，否则，返回原来的参数值。

脚注

98. 参考“库的展望”（7.13.2）。

99. 在使用 7 位 ASCII 字符集的实现下，打印字符是指值在 0x20（空格）和 0x7E（波浪符）之间的那些字符；控制字符是指值在 0（NULL）和 0x1F（US）之间的那些字符和字符 0x7F（DEL）。

2.3 <ctype.h> 的使用

<ctype.h> 中声明的函数可以用来对字符进行判断和转换，这些字符由 fgetc、getc、getchar 等函数（都在 <stdio.h> 中声明）读入。如果在判断它之前存储了一个这样的值，就要声明这个数据对象为整型。如果存储的是字符类型，那么就会丢失一些有用信息。用户可能会把文件结束指示误认为一个有效的字符。或者可能会把一个有效的字符代码转换为一个负值，这是不可接受的。

如果参数值不是上述几个函数读入的，那么一定要谨慎。这些函数只对 <stdio.h> 中定义的 EOF 值和 *unsigned char* 类型可以代表的值正确工作。当基本 C 字符集中的字符表示为 *char* 类型的时候，它们为正值，但其他字符可能不是这样。

对字符进行分类并不像看上去那么简单。首先，必须理解各种类；然后还要知道某个分类体系下的所有常用字符；同时要了解实现把那些不常用的字符隐藏在了什么位置；还需要了解把不同的字符集移植到一个具体实现上时，所有的一切是怎样改变的；最后，还要知道当程序改变区域设置的时候，这些分类是怎么改变的。

2

字符类别

字符分类函数定义的字符类别有以下几种。

- 数字——一个 '0' 到 '9' 之间的十进制数。
- 十六进制数字——数字，或者字母表的前 6 个字母 'A' 到 'F' 或 'a' 到 'f'。
- 小写字母——字母 a 到 z 中的一个，在 "C" 区域设置外可能会加上其他的字符。
- 大写字母——字母 'A' 到 'Z' 中的一个，在 "C" 区域设置外可能会加上其他的字符。
- 字母——小写字母或者大写字母，在 "C" 区域设置外可能会加上其他的字符。
- 字母数字——字母或者数字。
- 图形字符——占据一个打印位置，输出到显示设备时可见的字符。
- 标点符号——非字母数字的图形字符，至少包括表示 C 源程序文本的 29 个符号。
- 打印字符——图形字符或者空格符 ' '。
- 空格——空格字符 ' ' 和 5 个标准的运动控制字符（换页符 *FF*、换行符 *NL*、回车符 *CR*、水平制表符 *HT*、垂直制表符 *VT*）。在 "C" 区域设置外可能会加上其他的字符 。
- 控制字符——是 5 个标准的运动字符、退格符 *BS* 和警报符 *BEL*，加上其他可能的字符中的一个字符。

这些分类中有两类即使在 "C" 区域设置中也是没有边界的。实现可以定义任意数量的附加的标点或者控制字符。例如，在 ASCII 中，标点也包括像 '@' 和 '$' 这样的字符。控制字符包括编码为十进制的 1 和 31 之间的所有字符，和编码为 127 的删除符。

图 2-1 摘自 Plauger 和 Brodie 的 *Standard C*。它说明了字符分类函数之间的联系。圆角矩形中的所有字符都属于基本 C 字符集。这是那些用来表示任意的 C 源文件的字符。C 标准要求每个执行字符集都要包含所有的这些字符和编码为 0 的空字符。

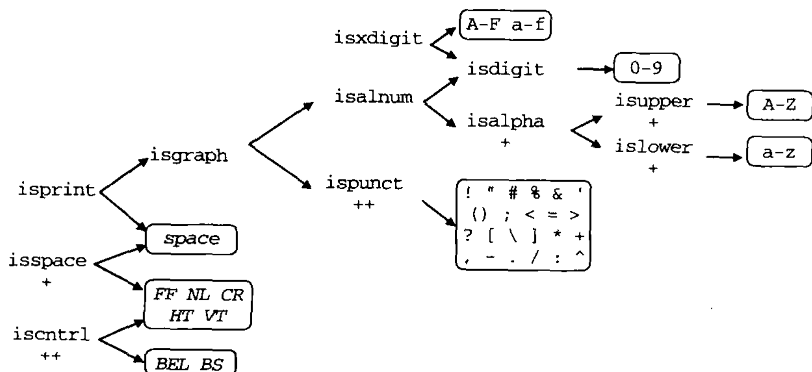


图 2-1
字符类别

函数名下有一个加号表明这个函数在 "C" 区域设置外的区域设置中可以表示附加的字符。两个加号表明这个函数即使在 "C" 区域设置下也可以表示附加的字符。

执行字符集也可以包含其他分类下的字符。但同一个字符只能位于图表中的一个位置。如果是小写字母，那么它也可以通过继承而属于多个类。但一个字符不能既是标点符号又是控制字符。

就像从图中看到的那样，几乎所有的函数都能在区域设置变动的程序中改变行为，只有 isdigit 和 isxdigit 保持不变。如果你的代码想处理区域设置语言，这是个好消息，你就可以使用这两个函数。例如，区域设置会改变 islower，来判别所有额外的小写字母。

区域设置改变时

然而，如果想让你的代码不受区域设置的限制，那么编程的时候就要更加谨慎了。使用字符分类函数来增加任何测试，以剔除任意的不在分类中的字符。或者，在程序改变它的区域设置为非 "C" 区域设置之前，去掉所有独立于区域设置的判别代码，使它们不会影响程序结果。

如果这些方法都不可行，你就可能必须把区域设置的一部分或者全部恢复为某个地区的习惯，可以参考 6.3 节。

最重要的是标准 C 开创了一个新的时代。现在可以更加容易地编写出适合世界各种文化的代码，这点是非常好的。但是现在写代码之前必须作更多的计划。如果它最后作为一个国际应用程序使用，某一天它可能会处理到早期的 C 程序员意想不到的字符。字符分类函数包含了这个问题，可以帮助处理这个问题，而且指出那些可以改变的东西。

下面，我会对 <ctype.h> 中声明的每个函数分别做一些说明。

isalnum

isalnum —— “Alnum” 是 “alphanumeric”（字母数字）的缩写。程序寻找名字的常见做法是，要求每个名字以字母开头，但是后面可以是数字或者字母。经常使用这个函数来检查名字的尾部字符。

isalpha

isalpha—— “Alpha” 是 “alphabetic”（字母）的缩写，不区分大小写。可以使用这个函数测试本地字母表中的字母。对于 "C" 区域设置来说，字母表总是包括我们熟悉的 26 个英文字母，包括大小写形式。

iscntrl

iscntrl——有些程序员认为这个函数和函数 isprint 的功能互补。当然，这两个函数识别两个不相交的集合，但是，这两个集合并不一定包含所有的字符。用这种方式使用 iscntrl 函数的程序在遇到生僻字符的时候会发生错误。

在使用这个函数的时候一定要小心。只有以下 7 个控制字符在所有的区域设置下有统一的行为——报警、退格、回车、换页、水平制表符、换行和垂直制表符。一个作了其他假设的程序应该用一个显著的注释记录下这些假设。

isdigit isdigit——这是各个区域设置中最稳定的函数之一。它只和基本 C 字符集中的 10 个十进制数字相对应，而不用管是什么区域设置。(一些字母表为各种各样的数字提供了额外的字符。）不仅如此，我们也可以保证这十个数字的编码也总是具有连续的值，就像我们通常的习惯做法那样（没有溢出检查）：

```
for(value = 0; isdigit(*s);++s)
    value = value * 10 + (*s -'0');
```

知道了这一点，就可以依靠这种习惯用法来简化那些实现数字转化的代码，并提高它们的运行速度。

isgraph isgraph——用来判别那些打印时可以显示的字符。该函数在修改区域设置的时候会改变它的行为。

islower islower——小写字母的组成在不同的区域设置中有相当大的区别。使用这个函数来确保你能识别所有的小写字母。不要认为每一个小写字母都有对应的大写字母，或者每一个大写字母都有对应的小写字母，甚至也不能认为字母只有大写和小写两种形式。

isprint isprint——这个函数可以识别所有输出到打印机时占据一个打印位置的字符。

ispunct ispunct——记住，标点符号是一个没有界限的字符集，即使在 "C" 区域设置中也是如此。就像 C 标准所描述的那样，你最好认为标点符号属于图形字符，而不属于字母数字。

isspace isspace——这是一个重要的函数。有些库函数使用 isspace 来判定把哪些字符作为空格对待。在 "C" 区域设置中，这个函数用来识别输出到显示设备时，所有改变打印位置，却没有显示图形的字符。你应该认为 isspace 在任意的区域设置中都是测试空格的最好选择。

isupper isupper——和上面 islower 的说明相同，只不过功能相反。

isxdigit isxdigit——和 isdigit 一样，这个函数也不随着区域设置而改变。你可以专门使用它来识别十六进制数。但是，你不能假设字母的编码像数字编码那样是邻接的。为了可以在所有的区域设置上对十六进制数进行转换，你可以写：

```
#include <ctype.h>
#include <string.h>
...
    static const char xd[] =
        { "0123456789abcdefABCDEF");
    static const char xv[] =
        {0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
        10, 11, 12, 13, 14, 15,
        10, 11, 12, 13, 14, 15};

    for (value = 0; isxdigit(*s); ++s)
        value = (value << 4) + xv[strchr(xd, *s) - xd];
```

注意这些代码没有对溢出进行检查，因为那会使这些程序更复杂。

tolower tolower——这个函数把大写字母转化为小写字母。它也可以将那些没有大写形式的字母和大小写形式都没有的字母处理为小写字母。不要认为通过简单的加上或者减去一个常量就能把大写字母转换为对应的小写字母。虽然那恰好适用于 ASCII 和 EBCDIC 这两种常用的字符集，但是 C 标准并没有这样要求。

toupper toupper——这个函数把小写字母转换为大写字母。它和上面的 tolower 有相同的说明，只不过功能相反。

2.4 <ctype.h> 的实现

这里给出的实现沿用了传统的方法。一个转换表描述了执行字符集的特征。每一个函数把它的参数作为表的入口，然后这些函数再把选择的表元素和一个唯一的掩码比较来确定参数字符是否在这种正在讨论的类别中。

若一个转换表很大，那它就没有什么意义了。它的大小是由它包含的元素个数和每个元素的大小决定的。标准 C 定义了 3 种“字符”类型——*char*、*signed char* 和 *unsigned char*。这些类型一定要能表示执行字符集的所有字符。至少要用 8 位才能表示所有的字符。

取值范围 每个字符分类函数都接收一个 *int* 类型的参数，但是参数值都在一定的范围内。*unsigned char* 类型所能表示的所有值都是有效的，还有宏 EOF（<stdio.h> 中定义）所确定的值。大部分好的实现都把 EOF 的值定义为 -1。本书也不例外。所以转换表中元素的数目一定比字符类型所能表示的所有字符的数目多 1。

大部分 C 实现严格地使用 8 位来表示一个字符类型。因此，一个转换表一定要包含 257 个元素。然而，实现可以使用更多的位数。C 已经分别使用了 9 位、10 位、16 位甚至 32 位来表示字符类型。而一个转换表要表示 16 位字符类型表示的所有值是很难处理的，因为它要包含 65 537 个元素。

图 2-1 显示了 8 种不同的类别，因而转换表可以是一个 *unsigned char* 类型的数组。但是该图也说明了这些分类中实现者可以加入字符的 6 个地方（有加号）。这就说明这个表一定是一个 *short* 类型的数组。我们可以把这些增加的字符加入到现有的分类中。但是，至少有两个增补的字符集处于 "C" 区域设置之外：

- 函数 isalpha 可以识别 islower 和 isupper 都不能识别的字符。
- 函数 isspace 可以识别 iscntrl 和 isprint 都不能识别的字符。

我们必须在以下两种方法中作出选择，或者清除那些有奇异的字符和空格的区域设置，或者使转换表的每个元素足够大以容纳 10 个分类位。如果想支持拥有这些字母或者空格字符的区域设置，就要把转换表声明为一个 *short* 类型的数组。然而，如果想清除这些范围内的值，可以通过把转换表声明为 *unsigned char* 类型的数组来节省空间。因为实现的目标是最大程度的可移植，所以它采用了前者。

不过，有一点是不能忽略的。我一直在说一个 8 位的转换表的元素应该是 *unsigned char* 类型。不是所有的实现都用 2 的补码来表示整数。在其他的表示方法中，把一个有符号的负数转换为无符号数可能会改变低位的值。因此，在一个有符号值和一个无符号掩码之间进行按位与操作的时候可能会发生意想不到的结果。

目前为止，我们假设字符用 8 位（或者不高于 8 位）表示，也假设了一个程序能够承担起一个 514（或者不多于）字节的转换表的开销。为了正确地给出一些实际的代码，至少还要再做 3 个假设。

tolower
toupper

假设 1：字母转换函数 tolower 和 toupper 和这个组中的其他函数不同。它们不是简单地对参数进行分类，而是返回一个可能和参数字符不同的字符。因此假设它们应该使用和转换表相似的映射表来实现，并且这个映射表可以被其他函数共享。

ASCII 和
ISO 646

假设 2：执行字符集是在现代计算机上广泛使用的 ASCII。另一种不同的国际标准 ISO 646 和 ASCII 有相同的代码值和图示符，或者字符的可视形式。因此，ASCII 中的某些标点符号可以被 ISO 646 中的图示符代替。这就是欧洲人引入带重音的字符的方法，例如 À 和 ê，而且不用多于 7 位的编码。

这个实现对那些没有把标点重新定义为字母的 ISO 646 的所有变化形式也是兼容的。然而对它做些修改来适应 ISO 646 的其他形式是很容易的。同样你可以很容易地修改其他字符集。IBM 的 EBCDIC 也要求对表项作些简单改动。只要保证表的项和 C 翻译器产生的字符常量（例如 'a'）相符合就可以了。

共享库 假设3：库可以使用指向表的指针的可写的静态存储空间。那只支持一种简单的情况，即翻译器必须包含C标准库中的代码。一旦包含在程序中，库的代码就和程序员提供的代码具有相同的行为。然而，那些可以运行多个程序的实现经常可以从共享库中受益。C标准库中的所有代码都占据一个独立的内存空间。一个链接好并在这个环境中运行的C程序把控制权转移给共享库中的函数，而不是把库中的代码复制到自己的程序中。这样做的一个明显的好处是程序更小而且链接得更快。

可写的静态存储空间 当一个或者多个函数需要维持一个属于库私有的可写的静态数据对象时，一个不太明显的弊端就浮现出来了。我们不能在不同的程序，或者在同一个程序的不同的控制线程之间共享一个相同的数据对象。我们需要为每一个程序或者线程分配一个唯一的可写的静态数据对象，并且要用合适的初值对它进行初始化。

不幸的是，还没有一个常规的方法来实现这种功能。操作系统和连接器使用特定的机器系统使共享库工作。有些直接禁用可写的静态存储空间，其他的则要求启用特定的机器系统来建立和访问可写的静态存储空间。因此，我们必须用一种特殊的方式来编写代码。

如果字符分类函数需要适应区域设置的改变，它们就需要可写的静态存储空间。一种方法是当区域设置改变时重新写表；另一个更好的方式是改变指向各个（只读）表的指针，那就加快了区域设置改变的速度，同时也大大减少了那些可能需要特别处理的可写的存储空间。

这种表述忽略了一个潜在的问题，这个问题跟库的可写的静态存储空间相联系。我尽可能少用可写的静态存储空间，也尽力注意那些必须引入可写的静态数据对象的代码，但是访问这些存储空间时并没有做特殊的标记。

头文件 <ctype.h> 图2-2显示了文件ctype.h。<ctype.h>中声明的函数的代码都是围绕着3个转换表构建的。3个可写的指针一直指向对应于当前区域设置的表。注意，每一个函数都有一个对应的宏。那些定义了分类位的宏名都很晦涩难懂，这有助于节省表示的空间。在很多实现中，它也加速了对标准头文件的处理。

isalnum 等函数 这些函数的代码看起来很像宏。图2-3（isalnum.c）到图2-15（toupper.c）给出了这些函数的代码。

_Tolower _Toupper 图2-16显示了文件xtolower.c。它定义了指针_Tolower的初始值，也给了伴随tolower的转换表的ASCII版本。相似地，图2-17显示了文件xtoupper.c，它定义了指针_Toupper的初始值和伴随toupper的转换表的ASCII版本。

图 2-2
ctype.h

```
/* ctype.h standard header */
#ifndef _CTYPE
#define _CTYPE
        /* _Ctype code bits */
#define _XA    0x200 /* extra alphabetic */
#define _XS    0x100 /* extra space */
#define _BB    0x80 /* BEL, BS, etc. */
#define _CN    0x40 /* CR, FF, HT, NL, VT */
#define _DI    0x20 /* '0'-'9' */
#define _LO    0x10 /* 'a'-'z' * /
#define _PU    0x08 /* punctuation */
#define _SP    0x04 /* space */
#define _UP    0x02 /* 'A'-'Z' */
#define _XD    0x01 /* '0'-'9', 'A'-'F', 'a'-'f' */
       /* declarations */
int isalnum(int), isalpha(int), iscntrl(int), isdigit(int);
int isgraph(int), islower(int), isprint(int), ispunct(int);
int isspace(int), isupper(int), isxdigit(int);
int tolower(int), toupper(int);
extern const short *_Ctype, *_Tolower, *_Toupper;
       /* macro overrides */
#define isalnum(c) (_Ctype[(int)(c)] & (_DI|_LO|_UP|_XA))
#define isalpha(c) (_Ctype[(int)(c)] & (_LO|_UP|_XA))
#define iscntrl(c) (_Ctype[(int)(c)] & (_BB|_CN))
#define isdigit(c) (_Ctype[(int)(c)] & _DI)
#define isgraph(c) (_Ctype[(int)(c)] & (_DI|_LO|_PU|_UP|_XA))
#define islower(c) (_Ctype[(int)(c)] & _LO)
#define isprint(c) \
    (_Ctype[(int)(c)] & (_DI|_LO|_PU|_SP|_UP|_XA))
#define ispunct(c) (_Ctype[(int)(c)] & _PU)
#define isspace(c) (_Ctype[(int)(c)] &(_CN|_SP |_XS))
#define isupper(c) (_Ctype[(int)(c)] & _UP)
#define isxdigit(c) (_Ctype[(int)(c)] & _XD)
#define tolower(c) _Tolower[(int)(c)]
#define toupper(c) _Toupper[(int)(c)]
#endif
```

图 2-3
isalnum.c

```
/* isalnum function */
#include <ctype.h>

int (isalnum)(int c)
    {                          /* test for alphanumeric character */
    return (_Ctype[c] & (_DI |_LO |_UP|_XA));
    }
```

图 2-4 isalpha.c

```
/* isalpha function */
#include <ctype.h>

int (isalpha)(int c)
    {                                   /* test for alphabetic character */
    return (_Ctype[c] & (_LO|_UP|_XA));
    }                                                                     □
```

图 2-5 iscntrl.c

```
/* iscntrl function */
#include <ctype.h>

int (iscntrl)(int c)
    {                                      /* test for control character */
    return (_Ctype[c] & (_BB|_CN));
    }                                                                     □
```

图 2-6 isdigit.c

```
/* isdigit function */
#include <ctype.h>

int (isdigit)(int c)
    {                                                  /* test for digit */
    return (_Ctype[c] & _DI);
    }                                                                     □
```

图 2-7 isgraph.c

```
/* isgraph function */
#include <ctype.h>

int (isgraph)(int c)
    {                                      /* test for graphic character */
    return (_Ctype[c] & (_DI|_LO|_PU|_UP|_XA));
    }                                                                     □
```

图 2-8 islower.c

```
/* islower function */
#include <ctype.h>

int (islower)(int c)
    {                                    /* test for lowercase character */
    return (_Ctype[c] & _LO);
    }                                                                     □
```

图 2-9 isprint.c

```
/* isprint function */
#include <ctype.h>

int (isprint)(int c)
    {                                    /* test for printable character */
    return (_Ctype[c] & (_DI|_LO|_PU|_SP|_UP|_XA));
    }                                                                     □
```

图 2-10 ispunct.c

```
/* ispunct function */
#include <ctype.h>

int (ispunct)(int c)
    {                                   /* test for punctuation character */
    return (_Ctype[c] & _PU);
    }
```

图 2-11 isspace.c

```
/* isspace function */
#include <ctype.h>

int (isspace)(int c)
    {                                       /* test for spacing character */
    return (_Ctype[c] & (_CN|_SP|_XS));
    }
```

图 2-12 isupper.c

```
/* isupper function */
#include <ctype.h>

int (isupper)(int c)
    {                                     /* test for uppercase character */
    return (_Ctype[c] & _UP);
    }
```

图 2-13 isxdigit.c

```
/* isxdigit function */
#include <ctype.h>

int (isxdigit)(int c)
    {                                       /* test for hexadecimal digit */
    return (_Ctype[c] & _XD);
    }
```

图 2-14 tolower.c

```
/* tolower function */
#include <ctype.h>

int (tolower)(int c)
    {                                /* convert to lowercase character */
    return (_Tolower[c]);
    }
```

图 2-15 toupper.c

```
/* toupper function */
#include <ctype.h>

int (toupper)(int c)
    {                                /* convert to uppercase character */
    return (_Toupper[c]);
    }
```

图 2-16 xtolower.c

```
/* _Tolower conversion table -- ASCII version */
#include <ctype.h>
#include <limits.h>
#include <stdio.h>
#if EOF != -1 || UCHAR_MAX != 255
#error WRONG TOLOWER TABLE
#endif

        /* static data */
static const short tolow_tab[257] = {EOF,
 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
 0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
 0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f,
 0x20, 0x21, 0x22, 0x23, 0x24, 0x25, 0x26, 0x27,
 0x28, 0x29, 0x2a, 0x2b, 0x2c, 0x2d, 0x2e, 0x2f,
 0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37,
 0x38, 0x39, 0x3a, 0x3b, 0x3c, 0x3d, 0x3e, 0x3f,
 0x40,  'a',  'b',  'c',  'd',  'e',  'f',  'g',
  'h',  'i',  'j',  'k',  'l',  'm',  'n',  'o',
  'p',  'q',  'r',  's',  't',  'u',  'v',  'w',
  'x',  'y',  'z', 0x5b, 0x5c, 0x5d, 0x5e, 0x5f,
 0x60,  'a',  'b',  'c',  'd',  'e',  'f',  'g',
  'h',  'i',  'j',  'k',  'l',  'm',  'n',  'o',
  'p',  'q',  'r',  's',  't',  'u',  'v''  'w',
  'x',  'y',  'z', 0x7b, 0x7c, 0x7d, 0x7e, 0x7f,

 0x80, 0x81, 0x82, 0x83, 0x84, 0x85, 0x86, 0x87,
 0x88, 0x89, 0x8a, 0x8b, 0x8c, 0x8d, 0x8e, 0x8f,
 0x90, 0x91, 0x92, 0x93, 0x94, 0x95, 0x96, 0x97,
 0x98, 0x99, 0x9a, 0x9b, 0x9c, 0x9d, 0x9e, 0x9f,
 0xa0, 0xa1, 0xa2, 0xa3, 0xa4, 0xa5, 0xa6, 0xa7,
 0xa8, 0xa9, 0xaa, 0xab, 0xac, 0xad, 0xae, 0xaf,
 0xb0, 0xb1, 0xb2, 0xb3, 0xb4, 0xb5, 0xb6, 0xb7,
 0xb8, 0xb9, 0xba, 0xbb, 0xbc, 0xbd, 0xbe, 0xbf,
 0xc0, 0xc1, 0xc2, 0xc3, 0xc4, 0xc5, 0xc6, 0xc7,
 0xc8, 0xc9, 0xca, 0xcb, 0xcc, 0xcd, 0xce, 0xcf,
 0xd0, 0xd1, 0xd2, 0xd3, 0xd4, 0xd5, 0xd6, 0xd7,
 0xd8, 0xd9, 0xda, 0xdb, 0xdc, 0xdd, 0xde, 0xdf,
 0xe0, 0xe1, 0xe2, 0xe3, 0xe4, 0xe5, 0xe6, 0xe7,
 0xe8, 0xe9, 0xea, 0xeb, 0xec, 0xed, 0xee, 0xef,
 0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7,
 0xf8, 0xf9, 0xfa, 0xfb, 0xfc, 0xfd, 0xfe, 0xff};

const short *_Tolower = &tolow_tab[1];
```
□

注意 #error 指令的使用，它保证了只有在假设成立的情况下代码才被正确地翻译。<limits.h> 中定义的宏 UCHAR_MAX 给出了 *unsigned char* 类型所能表示的最大值。

图 2-17
xtoupper.c

```
/* _Toupper conversion table -- ASCII version */
#include <ctype.h>
#include <limits.h>
#include <stdio.h>
#if EOF != -1 || UCHAR_MAX != 255
#error WRONG TOUPPER TABLE
#endif

       /* static data */
static const short toup_tab[257] = {EOF,
 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
 0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
 0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f,
 0x20, 0x21, 0x22, 0x23, 0x24, 0x25, 0x26, 0x27,
 0x28, 0x29, 0x2a, 0x2b, 0x2c, 0x2d, 0x2e, 0x2f,
 0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37,
 0x38, 0x39, 0x3a, 0x3b, 0x3c, 0x3d, 0x3e, 0x3f,
 0x40,  'A',  'B',  'C',  'D',  'E',  'F',  'G',
  'H',  'I',  'J',  'K',  'L',  'M',  'N',  'O',
  'P',  'Q',  'R',  'S',  'T',  'U',  'V',  'W',
  'X',  'Y',  'Z', 0x5b, 0x5c, 0x5d, 0x5e, 0x5f,
 0x60,  'A',  'B',  'C',  'D',  'E',  'F',  'G',
  'H',  'I',  'J',  'K',  'L',  'M',  'N',  'O',
  'P',  'Q',  'R',  'S',  'T',  'U',  'V',  'W',
  'X',  'Y',  'Z', 0x7b, 0x7c, 0x7d, 0x7e, 0x7f,

 0x80, 0x81, 0x82, 0x83, 0x84, 0x85, 0x86, 0x87,
 0x88, 0x89, 0x8a, 0x8b, 0x8c, 0x8d, 0x8e, 0x8f,
 0x90, 0x91, 0x92, 0x93, 0x94, 0x95, 0x96, 0x97,
 0x98, 0x99, 0x9a, 0x9b, 0x9c, 0x9d, 0x9e, 0x9f,
 0xa0, 0xa1, 0xa2, 0xa3, 0xa4, 0xa5, 0xa6, 0xa7,
 0xa8, 0xa9, 0xaa, 0xab, 0xac, 0xad, 0xae, 0xaf,
 0xb0, 0xb1, 0xb2, 0xb3, 0xb4, 0xb5, 0xb6, 0xb7,
 0xb8, 0xb9, 0xba, 0xbb, 0xbc, 0xbd, 0xbe, 0xbf,
 0xc0, 0xc1, 0xc2, 0xc3, 0xc4, 0xc5, 0xc6, 0xc7,
 0xc8, 0xc9, 0xca, 0xcb, 0xcc, 0xcd, 0xce, 0xcf,
 0xd0, 0xd1, 0xd2, 0xd3, 0xd4, 0xd5, 0xd6, 0xd7,
 0xd8, 0xd9, 0xda, 0xdb, 0xdc, 0xdd, 0xde, 0xdf,
 0xe0, 0xe1, 0xe2, 0xe3, 0xe4, 0xe5, 0xe6, 0xe7,
 0xe8, 0xe9, 0xea, 0xeb, 0xec, 0xed, 0xee, 0xef,
 0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7,
 0xf8, 0xf9, 0xfa, 0xfb, 0xfc, 0xfd, 0xfe, 0xff};

const short *_Toupper = &toup_tab[1];
```
□

数据对象 _Ctype

图 2-18 给出了文件 xctype.c。所有的字符分类函数都共用一个 _Ctype 指向的转换表，这个文件就定义了这个表和指针。

图 2-18
xctype.c

```
/* _Ctype conversion table -- ASCII version */
#include <ctype.h>
#include <limits.h>
#include <stdio.h>
#if EOF != -1 || UCHAR_MAX != 255
#error WRONG CTYPE TABLE
#endif

        /* macros */
#define XDI (_DI|_XD)
#define XLO (_LO|_XD)
#define XUP (_UP|_XD)
         /* static data */
static const short ctyp_tab[257] = {0, /* EOF */
_BB, _BB, _BB, _BB, _BB, _BB, _BB, _BB,
_BB, _CN, _CN, _CN, _CN, _CN, _BB, _BB,
_BB, _BB, _BB, _BB, _BB, _BB, _BB, _BB,
_BB, _BB, _BB, _BB, _BB, _BB, _BB, _BB,
_SP, _PU, _PU, _PU, _PU, _PU, _PU, _PU,
_PU, _PU, _PU, _PU, _PU, _PU, _PU, _PU,
XDI, XDI, XDI, XDI, XDI, XDI, XDI, XDI,
XDI, XDI, _PU, _PU, _PU, _PU, _PU, _PU,
_PU, XUP, XUP, XUP, XUP, XUP, XUP, _UP,
_UP, _UP, _UP, _UP, _UP, _UP, _UP, _UP,
_UP, _UP, _UP, _UP, _UP, _UP, _UP, _UP,
_UP, _UP, _UP, _PU, _PU, _PU, _PU, _PU,
_PU, XLO, XLO, XLO, XLO, XLO, XLO, _LO,
_LO, _LO, _LO, _LO, _LO, _LO, _LO, _LO,
_LO, _LO, _LO, _LO, _LO, _LO, _LO, _LO,
_LO, _LO, _LO, _PU, _PU, _PU, _PU, _BB,
};                                         /* rest all match mothing */

const short *_Ctype = &ctyp_tab[1];
```

2.5 <ctype.h> 的测试

只有 <ctype.h> 声明的函数接收的参数值都有效时，对这些函数进行测试才有意义。最好既能测试这些函数又能测试隐藏这些函数的宏。当然，那就超出了只测试 <ctype.h> 外部特性的范围。这样的双重测试可以查找头文件和它的函数内部工作方式的漏洞。然而，这是一个函数和宏都很重要的实例。我们希望它们都能够按照预想的方式工作。

我们也可以从某些附加的信息中受益——比如按照编码值递增的顺序，显示出各种可打印类别的字符。那可以确保所有我们想看到的字符匹配，而且又没有多余的东西。它也显示了 "C" 区域设置允许的所有附加字符，例如附加的标点符号。同时，它也让我们看到了一个类别中各个字符的整理顺序。

图 2-19 显示了测试程序 tctype.c。这个程序显示了几种字符分类，然后测试了函数和它们的屏蔽宏。注意在第二个测试集合中函数名附近使用了括号。这跟我定义库中的每个可视函数所用的技巧相同。使用圆括号，那些带参数的宏就不会掩盖头文件前面的部分真正的函数的声明。如果执行字符集是 ASCII，程序的输出是：

```
ispunct: !"#$%&'()*+, -./: ;<=>?@ [\]^_'{|}~
isdigit: 0123456789
islower: abcdefghijklmnopqrstuvwxyz
isupper: ABCDEFGHIJKLMNOPQRSTUVWXYZ
isalpha: ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
isalnum: 0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrs
tuvwxyz
SUCCESS testing <ctype.h>
```

注意到，和 isalnum 匹配的那部分字符在这里被折叠，那是因为本书的页面还没有宽到可以把它们放在一行来显示。这行字符在一个屏幕足够大的电脑上不会被折叠。

图 2-19 tctype.c

```
/* test ctype functions and macros */
#include <assert.h>
#include <ctype.h>
#include <limits.h>
#include <stdio.h>

static void prclass(const char *name, int (*fn)(int))
    {                   /* display a printable character class    */
    int c;

    fputs(name, stdout);
    fputs(": ", stdout);
    for (c = EOF; c <= UCHAR_MAX; ++c)
        if ((*fn)(c))
            fputc(c, stdout);
    fputs("\n",stdout);
    }

int main()
    {                           /* test both macros and functions */
    char *s;
    int c;

        /* display printable classes */
    prclass("ispunct", &ispunct);
    prclass("isdigit", &isdigit);
    prclass("islower", &islower);
    prclass("isupper", &isupper);
    prclass("isalpha", &isalpha);
    prclass("isalnum", &isalnum);
        /* test macros for required characters */
    for (s = "0123456789"; *s; ++s)
        assert(isdigit(*s) && isxdigit(*s));
    for (s = "abcdefABCDEF"; *s; ++s)
        assert(isxdigit(*s));
    for (s = "abcdefghijklmnopqrstuvwxyz"; *s; ++s)
        assert(islower(*s));
    for (s = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"; *s; ++s)
        assert(isupper(*s));
    for (s = "!\"#%&'();<=>?[\\]*+,-./:^_{|}~"; *s; ++s)
        assert(ispunct(*s));
    for (s = "\f\n\r\t\v"; *s; ++s)
        assert(isspace(*s) && iscntrl(*s));
    assert(isspace('') && isprint(''));
    assert(iscntrl('\a') && iscntrl('\b'));
        /* test macros for all valid codes */
```

图 2-19
（续）

```
    for (c = EOF; c <= UCHAR_MAX; ++c)
        {                           /* test for proper class membership */
        if (isdigit(c))
            assert(isalnum(c));
        if (isupper(c))
            assert(isalpha(c));
        if (islower(c))
            assert(isalpha(c));
        if (isalpha(c))
            assert(isalnum(c) && !isdigit(c));
        if (isalnum(c))
            assert(isgraph(c) && !ispunct(c));
        if (ispunct(c))
            assert(isgraph(c));
        if (isgraph(c))
            assert(isprint(c));
        if (isspace(c))
            assert(c == '' || !isprint(c));
        if (iscntrl(c))
            assert(!isalnum(c));
        }
        /* test functions for required characters */
    for (s = "0123456789"; *s; ++s)
        assert((isdigit)(*s) && (isxdigit)(*s));
    for (s = "abcdefABCDEF"; *s; ++s)
        assert((isxdigit)(*s));
    for (s = "abcdefghijklmnopqrstuvwxyz"; *s; ++s)
        assert((islower)(*s));
    for (s = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"; *s; ++s)
        assert((isupper)(*s));
    for (s = "!\"#%&'();<=>?[\\]*+,-./:^-{|}~"; *s; ++s)
        assert((ispunct)(*s));
    for (s = "\f\n\r\t\v"; *s; ++s)
        assert((isspace)(*s) && (iscntrl)(*s));
    assert((isspace)('') && (isprint)(''));
    assert((iscntrl)('\a') && (iscntrl)('\b'));
        /* test functions for all valid codes */
    for (c = EOF; c <= UCHAR_MAX; ++c)
        {                           /* test for proper class membership */
        if ((isdigit)(c))
            assert((isalnum)(c));
        if ((isupper)(c))
            assert((isalpha)(c));
        if ((islower)(c))
            assert((isalpha)(c));
        if ((isalpha)(c))
            assert((isalnum)(c) && !(isdigit)(c));
        if (isalnum(c))
            assert((isgraph)(c) && !(ispunct)(c));
        if ((ispunct)(c))
            assert((isgraph)(c));
        if ((isgraph)(c))
            assert((isprint)(c));
        if ((isspace)(c))
            assert(c == '' || !(isprint)(c));
        if ((iscntrl)(c))
            assert(!(isalnum)(c));
        }
    puts("SUCCESS testing <ctype.h>");
    return (0);
    }
```
□

2.6 参考文献

近年来，人们对字符集的关注越来越多。商业国际化要求计算机可以支持更加丰富的字符集合，而不是传统的只能表示英文（和 C）。很多计算机销售商开始对标准的 8 位字节所能表示的 256 个编码进行赋值。然而，那些保守的人仍然坚持那个可以用 7 位编码的 128 个或者更少的字符的集合。很多实现遵循以下两个标准：

- *ANSI Standard X3.4-1968* (New York: American National Standards Institute, 1989). 这个标准定义了 ASCII 字符集合，该集合使用 7 位编码表示字符，在现代计算机中被广泛使用。
- *ISO Standard 646:1983* (Geneva: International Standards Organization, 1983). 这是 7 位字符编码的国际标准。

2.7 习题

2.1 列出对下面字符串中的每个字符返回非零值的所有的字符分类函数：

```
"Hello,world!\n"
```

2.2 修改 `<ctype.h>` 中声明的函数，使它能够对任意的参数值正确地工作。对一个超出范围的参数值可以采用和 EOF 值相同的处理方法。至少描述两种报告参数越界的错误方法。

2.3 C 中的名字以字母开头，后面可以跟任意多个字母、数字或者下划线。编写函数 `size_t idlen(const char *s)`，该函数返回的是以 s 开头的构成标识符的字符数量。如果没有以 s 开头的标志符，该函数返回 0。

2.4 编写函数 `size_t detab(char *dest, const char *src)`，该函数从地址 src 复制一个以空字符结束的字符串到 dest，并将源字符串中的每个水平制表符替换成 1 ~ 4 个空格。假设制表符 4 列一个，一个可打印字符占一列。其他的影响打印位置的字符是退格符、回车符和换行符。返回 dest 中新串的长度。

2.5 当区域设置改变为非 "C" 时，你在习题 2.3 中编写的函数 idlen 是否需要修改才能正确地工作？如果是，给出修改后的版本，如果不是，说明理由。

2.6 当区域设置改变为非 "C" 时，你在习题 2.4 中写出的函数 detab 是否需要修改才能正确地工作？如果是，给出修改后的版本，如果不是，说明理由。

2.7 [**难**] 假设你希望实现一个可以共享的库。描述一下在以下每种机制下你会怎样修改本章的代码：

- ❐ 通过命令，翻译器可以把库中所有的可写的静态存储空间复制为每个使用该共享库的过程的一个部分。
- ❐ 可以向 <libstat.h> 声明的结构 _Lib_stat 中添加域，也可以为文件 libstat.c 中该结构的定义添加初始值。
- ❐ 和上面相同，可以向 <libstat.h> 声明的结构 _Lib_stat 中添加域，但只能通过指向该结构的指针 _p 来访问它，这个指针也由 <libstat.h> 声明。
- ❐ 可以向 <libstat.h> 声明的结构 _Lib_stat 中添加域，而且只能通过一个指向该结构的指针来访问它，这个指针由函数调用 _FP() 返回，函数 _FP 也由 <libstat.h> 声明。

2.8 [**难**] 一个*多线程*的环境支持一个或者多个控制台控制的线程，这些线程使用相同的静态存储空间。每一个线程独立地使用动态存储空间（存储类别为 auto（自动）或者 register（寄存器））。你要实现线程的*原子*（atomic）库——在这个库中，所有的函数都不会因为另一个线程改变库的静态存储空间的状态而改变它们的行为，或者行为出错。你可以通过在每一个对库的静态存储空间的访问周围加上一些*同步代码*（synchronization code），来保证它的安全。就像下面的代码：

```
_lock();
p = _Ctype;
_unlock();
```

说明怎样改动本章的代码才能使它安全地应用于多线程操作。那对性能有什么影响？在多线程操作中，你怎样才能既提高性能又能保证代码的安全性？

2.9 [**很难**] 修改 <ctype.h> 中定义的宏，使它对任意的参数值都能正确地工作。对超出范围的参数值可以采用和 EOF 的值相同的处理方法。

第3章

<errno.h>

3

3.1 背景知识

如果必须要指出C标准中最不被人们喜欢的一部分，那它就在眼前。没有人喜欢errno或者它所指示的机制。我想不起有谁支持这种报告错误的方法，在制定C标准的委员会X3J11的二十多次会议上也是如此。这些年来也有人提出了几种可供选择的方案，至少有一部分人主张抛弃errno。可它至今仍然存在。

C标准甚至扩充了现有的机制。头文件<errno.h>是该委员会的发明。我们想让库中的每一个函数和数据对象都在某一个标准头文件中声明。我们给了errno一个自己的头文件来把它分离出来。我们甚至添加了一些说明，希望澄清C语言中不明确的地方。

一个在致力于扩展和改进C的群组中一直讨论的主题是怎样驯服errno，或者怎样摆脱它。到目前为止还没有一个明确的答案，这也许会告诉你点什么东西。关于报告和处理错误方面，没有什么简单的答案。

历史 C诞生于UNIX之下。该操作系统为了清晰和简单而定义了新的标准。用户程序和操作系统内核之间的接口特别清晰。你可以指定一个系统调用号和几个操作数。早期UNIX的40多个系统调用在那几十年中在数目上翻了一番多，但是和其他具有相当功能的系统相比，这还是比较少的。UNIX系统调用的操作数大多数都是标量——整数或者指针。它们同样都很少。

UNIX的每一个实现对错误的系统调用都采用一种简单的指示方法。用汇编语言实现，通常使用条件编码的进位标志来测试。如果进位标志被清零，系统调用就成功了。要求的所有结果都返回到机器寄存器或者程序的一个结构中。（要指定该结构的地址作为系统调用的一个参数。）如果进位标志不是零，系统调用就出错了。一个机器寄存器记录了一个很小的正数用来说明错误的性质。

C中的错误处理

那种方法对汇编来说非常好，但对于用C编写的程序来说就不是那么好了。可以编写一个C可调用的函数库，其中每个函数对应于一个系统调用。当执行一个特定的系统调用的时候，我们希望那个对应的函数的返回值为我们想要的结果。这样做虽然可以，但这会让以容易测试的方式报告错误变得很难。还有一种选择，可以让每个函数返回一个指示成功或者失败的值。但那样做，又很难从一个成功的系统调用中得到你想要的结果。

一个通常可行的做法是取两者的优点。对一个一般的系统调用来说，可以定义一个和所有的有效结果都不相同的错误返回值。在这点上，空指针就是一个例子。值 -1 也可以在很多情况下被设为错误返回值，而不会和有效结果冲突。每个UNIX系统调用通常有一个这样的返回值来说明发生了某些形式的错误。

C可调用的函数不能做到的是准确地报告发生了什么错误。这就过多地限制了这个巧妙的方法。你从返回值能得到的所有信息就是是否有错误发生。必须从其他的地方来获得细节。

早期的UNIX程序员采用的“其他地方”是一个具有外部连接的数据对象。任何失败的系统调用都从内核中存储一个叫做errno的整型变量作为错误编码。它或者返回 -1，或者返回其他适当的无意义的值来说明一个错误。程序在大多数情况下都不在乎细节，错误就是错误。但是在需要了解细节的少数情况下，它也知道怎样得到这些额外的信息。它去访问errno来得到存储在那里的最后的错误编码。

所以，你最好早点访问errno。如果另一个系统调用也失败了，原来的错误编码就会被新的错误编码覆盖掉。你也只能在一个系统调用失败后再访问errno的值。一个成功的系统调用不会清空存储在那的值。它不是一个很好的机制，但它确实很实用。

过劳的机制

errno的第一个问题是它太方便了。人们便开始寻找它的更多用途。errno从一个为了增加UNIX系统调用数量的小窍门发展为C的一个制度。这时，它就开始过度劳累了。系统调用并不是大量错误发生的唯一地方。另一个公认的地方是计算一般数学函数的库部分。（见第7章。）

一些函数接受某些参数后产生的值过大而不能表示（例如 exp(1000.0)）；一些则产生的值过小而不能表示（例如 exp(-1000.0)）；一些函数对某些参数值没有定义（例如 sqrt(-1.0)）；而一些函数定义了，但是对参数值没有什么价值（例如 sin(1e30)）。

可以为每一个可能引起麻烦的函数引入一个或者多个错误编码。根据UNIX错误编码的命名习惯，对一个负数的平方根可以报告ESQRT。但是这样就没有了界限而且容易引起混乱。

数学错误

幸运的是，数学错误可以被归结为以下几类：

- 当一个结果在数值上太大而不能作为指定类型的浮点值表示的时候就会发生向上溢出；
- 当一个结果在数值上太小而不能作为指定类型的浮点值表示的时候就会发生向下溢出；
- 但一个结果没有位置容纳它的类型指示的有效位的时候就会发生有效值丢失；
- 当接受一个指定的参数值而产生的结果没有被定义的时候就会发生域错误。

UNIX 下一些不同的系统调用会产生相同的错误代码。相似地，一些不同的数学函数也会产生一个或者多个这样的错误（在使用浮点操作数的时候，这些错误甚至会在所有的数学操作符中产生）。实际上，仅使用两个错误代码就足够表示所有的数学错误：

- 当一个域错误发生时报告 EDOM;
- 当溢出发生时报告 ERANGE。

有效值丢失是一个不确定是否要报告的错误。一个程序员对有效值丢失的看法可能和另一个程序员完全不同。确实，对于计算过程这一部分而言，一个稳定的算法对一个严格的有效值丢失并不是很敏感。因此，有效值丢失是否应该被库报告是一个有争议的话题。

你可能已经知道下面要说什么了。数学库中可能发生的错误和系统调用的一样多。我们需要通过某种方式来报告数学库的错误。所以当手中已经有了一种机制时为什么要重新定义一个呢？通过在 errno 中存储 EDOM 和 ERANGE 来报告数学错误是 C 标准库的一个较早的、自然的演化。这种做法已经被 C 标准认可并包含在内。C 标准也清楚地说明了库函数中其他几个需要设置 errno 的地方。完整的清单是：

定义的错误

- <math.h> 中声明的很多函数都将 <errno.h> 中声明的宏 EDOM 和 ERANGE 的值存储在 errno 中。
- <stdlib.h> 中声明的一些函数把文本串转换为各类算术类型的值。这些函数的部分或者全部将 ERANGE 存储在 errno 中。
- <stdio.h> 中声明的一些函数改变文件中下一次读或者写操作的位置。这些函数会在 errno 中存储一个正值。这个值是实现定义的。本书的实现选择 EFPOS 作为 <errno.h> 中定义的宏名，该宏和那个值相对应。它不是一个常用的名字。
- <signal.h> 中声明的函数 signal，也在 errno 中存储一个正值。该值甚至也不是实现定义的——一个实现可以按照它的选择去做，但并不显示它做了什么。因为 signal 在各种实现中差别太大，所以在这个库中，我没有指定特定的错误编码。

3.2 C 标准的内容

<errno.h>

7.1.4 错误报告 <errno.h>

头文件 <errno.h> 定义了几个宏，这些宏都和错误状态的报告有关。

这些宏是：

```
EDOM
ERANGE
```

EDOM
ERANGE

它们展开为具有不同非零值的整数常量表达式，适合在 #if 预处理指令中使用。而宏

errno

errno

则展开为一个可以修改的 int 类型的左值[92]，它的值被某几个库函数设置为一个正的错误编号。无需说明 errno 是一个宏还是一个声明为具有外部连接的标识符。如果为了访问一个实际的对象掩盖了一个宏定义，或者一个程序定义了一个名字为 errno 的标识符，那么这种行为是未定义的。

errno 的值在程序启动时为零，但是它决不会被任何库函数设为零[93]。如果 errno 的使用没有记录在这个国际标准中的函数说明中，那么不管是否存在错误，errno 的值都可能被设置为非零。

以 E 和一个数字或者 E 和一个大写字母开头的附加宏的定义，也可能由实现来详细说明。

脚注

92. 宏 errno 不一定是一个对象的标识符，它也可能展开为一个函数调用返回的，可以修改的 lvalue（例如 *errno()）。
93. 因此，一个程序如果使用 errno 进行错误检查，应该在一个库函数调用之前把它设为零，然后，在下一个库函数调用之前查看它。当然，一个库函数可以先保存 errno 的值，然后再把它设为零，在函数返回之前，如果 errno 的值仍为零，只要让它恢复为原始值就可以了。
94. 参考“库的展望”（7.13.1）。

3.3 <errno.h> 的使用

C 标准关于可以报告的错误说得并不是很详细，它甚至对所有错误编码的值或者用来确定那些值的宏名说得更少。那是因为在各种实现中它们的用法有很大的不同。甚至不同的 UNIX 版本都定义不同的错误编码集合。

如果你为一个特定的系统写代码，你可能不得不学习它特定的错误编码集合。如果可能的话，列出头文件 <errno.h>。所有的错误编码都应该作为以 E 开头的宏在那里被定义。阅读所有你可以找到的关于错误编码细节的文档，然后准备做些试验。在这个领域，文档一向不是那么完整和准确。

如果想编写可移植的代码，一定要避免任何附加的错误编码的假设。只能依靠 C 标准中指定的 errno 属性，3.1 节中有完整列表。然而，几乎不需要很清楚地知道错误编码。C 标准的脚注 93（上面已给出）讲述了使用 errno 最安全的编码方式。在一个库函数调用之前把它设为 0，然后在下一个库调用前测试它的任何非零值：

```
#include <errno.h>
#include <math.h>
...
    errno = 0;
    y = sqrt(x);
    if (errno != 0)
        printf("invalid x: %e\n",x);
```

不论一个库函数是多么简单，决不要认为它不会对 errno 造成任何影响。这是一个相当令人头疼的方法。

3.4 <errno.h> 的实现

从表面来看， C 标准在这个领域对实现的要求很少，可以简单地把文件 errno.h 写为：

```
/* errno.h standard header */
#ifndef _ERRNO
#define _ERRNO

#define EDOM   1
#define ERANGE 2

extern int errno;

#endif
```

在某个库文件中，必须为数据对象添加一个定义：

```
int errno = 0;
```

唯一必须做的另外一件事是在库函数内适当的位置把像 EDOM 和 ERANGE 这样的值存储在 errno 中。还有比这更简单的吗？

这里有一个例子，在这个例子中这种实现显然是全部工作中最简单的部分。errno 会在两个微妙的方向制造麻烦——有时它的说明太模糊，有时它的说明又太清楚了。下面就是这方面的解释。

太多和太少

模糊性源于过去对 errno 的最初记录系统调用错误的用法。这种用法已经被 C 标准默认了。所有库函数都可以在 errno 中存储非零值。这些存储可以实现是因为函数可以执行一个或者多个失败的系统调用，或者因为某个库函数选择使用这种报告（错误）的方法。

所有能依赖的只有 C 标准中明确规定的那部分行为。调用 sqrt(-1.0) 就可以确定地知道 errno 包含 EDOM 的值。不管你是否相信，调用 fabs(x) 对 errno 都没有任何影响。没有一个库函数会把 0 存储到 errno 中。而其他任何值则都有可能。

规定太多主要影响了数学函数。通过清楚地说明 errno 必须设定的场合，C 标准干预了重要的优化部分。特别地，C 标准使编译器很难利用最新的浮点运算协处理器的优势。

像 Intel 80X87 系列和 Motorola MC68881 这样的芯片有很多相当复杂的指令。一些指令可以用内联代码计算一个数学函数的部分或者全部。一个好的编译器可以通过使用这些指令极大地提高计算速度。如果没有其他的东西，一个编译器可以为一个数学函数避免函数调用和返回的开销。

数学异常

当一个数学异常发生时，问题就出现了。这些数学协处理器独立地工作，而且想不停地运转。它们希望通过传送一个特别的、叫做 NaN（“Not a Number”的缩写）或者 Inf（“infinity”的缩写）的编码来记录一个错误。后面的操作保留这些特别的代码。我们可以在计算的最后来检查整个过程中是否有错误发生。

最好的选择是，这些协处理器用它们自己的状态码记录错误的发生。主处理器必须把协处理器的状态码复制过来才能检查是否有错误发生。这就阻止了流水线协处理器的全速运转。如果一个 C 程序必须在每一个数学异常中设置 errno，它就不能让协处理器以最高的速度来运行。

宏 errno

C 标准的脚注 92 提出了一种解决方法。C 标准并没有要求 errno 是一个真实的数据对象。它作为一个宏来定义，这个宏可以被扩展为一个*可更改的左值*——一个可以在赋值操作符（例如 '='）左边指定一个数据对象的任意的表达式。这就给了实现者相当大的空间。特别地，宏 errno 可以扩展为像 *_Erfun() 这样的表达式。每当程序想检查错误的时候，它就会调用一个函数来告诉程序去哪个地方找。

这样做有两个含义。首先，实现可能不想记录错误，它可以等到有人急切地寻找 errno 的时候才记录最新的错误编码。这就可能给实现足够的自由来让数学协处理器在大部分时间独立地工作（然而，翻译器可能被强迫使用这种方法）。

第二个含义是 errno 可以到处移动。函数在每次被调用时都可以返回一个不同的地址。这对实现共享库是一个很大的帮助。就像 2.4 节所说的，静态存储空间对共享库来说是一个麻烦的东西。用户程序可以任意改变的静态存储空间的情况更糟。errno 是 C 标准库在这方面唯一的创造。

即使作为一个宏，errno 仍然是体制中令人讨厌的一部分。任何程序都可以包含下面的代码：

```
y = sqrt(x);
if (errno == EDOM)
    ...
```

支持这种错误检测的需求严重地限制了实现对 sqrt 之类的函数的操作。因为任何库函数都可以修改 errno，所以程序员也不能得到很好的服务。这里我们得到的是一个让实现者和用户都很为难的机制。

参数编码

图 3-1 给出了 errno.h 的代码。它不像我在前面说明的代码那么简单。那是因为我想使它参数化。这种简单的形式一定要作些修改以适合每个管理库的操作系统。其他的库函数或者操作系统本身可能对错误编码的值有预定的概念。我们必须修改这个头文件来匹配或者容忍这种令人惊讶的无规则性。

图 3-1 errno.h

```
/* errno.h standard header */
#ifndef _ERRNO
#define _ERRNO
#ifndef _YVALS
#include <yvals.h>
#endif
        /* error codes */
#define EDOM    _EDOM
#define ERANGE _ERANGE
#define EFPOS   _EFPOS
    /* ADD YOURS HERE */
#define _NERR _ERRMAX                       /* one more than last code */
        /* declarations */
extern int errno;
#endif
```

使用 <errno.h> 的大部分代码都很在意一两个错误编码的值。就像 3.2 节中提到的那样，这些值随操作系统的改变而改变。一两个库函数需要知道错误编码的有效范围。这个取值范围也因操作系统而异。

在第一次实现这个库之后，我开始把它移植到各种环境下。我发现十几个文件必须在很小的地方修改。很快，我就被维护这十几个文件的各个版本打败了。

头文件 <yvals.h>

这就促使我引入一个叫做“内部标准头文件”的东西。一些标准头文件包含头文件 <yvals.h>（尖括号告诉编译器在其他标准头文件存储的地方找这个头文件，这可能会在某些系统上产生问题）。我把为实现库的可移植性必须修改的东西集中到这个文件中。

头文件 <errno.h> 以 <yvals.h> 中定义的其他宏为基础定义了自己的宏。这种分两步的过程是必要的，因为其他的头文件也包含 <yvals.h>。只有当程序包含了 <errno.h> 时，才能定义宏 ERANGE。

同时也要注意 <yvals.h> 的宏保护是在包含 <yvals.h> 的头文件中，而不是在 <yvals.h> 中。这是一个小的优化。因为几个标准头文件包含这个头文件时，它很可能在一个翻译单元中被多次请求。一旦 <yvals.h> 变为翻译单元的一部分，宏保护就会跳过 #include 预处理指令。这个头文件也就不会被重复地读入。

图 3-2
errno.c

```
/* errno storage */
#include <errno.h>
#undef errno

int errno = 0;
```

头文件 <yvals.h> 是各种值的大杂烩。附录 A 给出了各种常见的操作系统的头文件版本。这里只列出了 <yvals.h> 中影响 <errno.h> 的宏。这些值和装载在 Sun 工作站的 UNIX 系统下 Borland Turbo C++ 的编译器和 DEC VAX 下的 ULTRIX 的编译器相一致：

_EDOM
_ERANGE
_EFPOS
_ERRMAX

```
#define _EDOM   33
#define _ERANGE 34
#define _EFPOS  35
#define _ERRMAX 36
```

然而，请注意，这些值并不适用于所有的情况。

头文件 "yfuns.h"

我要强调的是，使这个库适用于一个给定的操作系统，仅仅有 <yvals.h> 是不够的。在这本书的后面，我会引入另一个叫做 "yfuns.h" 的头文件（见 12.4 节）。该头文件的作用与其相似但并不相同。甚至两个头文件也是不够的。C 标准库中的很多函数在不同的操作系统中参数化的差别太大，因此，它们也出现了不同的版本。你会在后面的章节中不时地看到它们。

图 3-2 显示了定义 errno 数据对象的文件 error.c。#undef 预处理指令只是将来 <errno.h> 改变的一个安全保证。

图 3-3
terrno.c

```
/* test errno macro */
#include <assert.h>
#include <errno.h>
#include <math.h>
#include <stdio.h>

int main()
    {                           /* test basic workings of errno */
    assert(errno == 0);
    perror("No error reported as");
    errno = ERANGE;
    assert(errno == ERANGE);
    perror("Range error reported as");
    errno = 0;
    assert(errno == 0);
    sqrt(-1.0);
    assert (errno == EDOM) ;
    perror("Domain error reported as");
    puts("SUCCESS testing <errno.h>");
    return(0);
    }
```

3.5 <errno.h> 的测试

图 3-3 显示了测试程序 terrno.c。它做的工作不多。C 标准对 <errno.h> 的属性描述得很少。terrno.c 主要用来保证一个程序可以把值存储到 errno 中并且能找回这些值。

为了便于查看，测试程序也显示了标准错误编码是如何在输出时出现的。<stdio.h> 中声明的函数 perror 写入一行文本到标准错误流。这个函数根据 errno 的内容确定文本行的最后一部分。如果一切顺利，运行 terrno.c 的可执行程序会显示以下输出：

```
No error reported as: no error
Range error reported as: range error
Domain error reported as: domain error
SUCCESS testing <errno.h>
```

我必须再次提醒：这个输出既来自标准错误流又来自标准输出流。虽然此处这种情况的可能性很小，但某些实现可能会重新排列这些文本行。

3.6 参考文献

David Stevenson, "A Proposed Standard for Binary Floating-Point Arithmetic," *Computer,* 14:3 (1981), pp. 51-62. *Computer* 杂志的这篇文章和同一期后面的一些文章解释了 IEEE 754 浮点数标准的很多方面。

Mark J. Rochkind, *Advanced UNIX Programming* (Englewood Cliffs, N.J.: Prentice Hall, Inc., 1985). Rochkind 描述了 errno 和它的错误编码起源的地方——UNIX 系统调用。

3.7 习题

3.1 列出你使用的 C 翻译器定义的错误编码。你能用一句话描述每个错误编码说明了什么错误吗？

3.2 对你使用的 C 翻译器定义的错误编码，为每一个编码设计一个可以使它发生的测试实例。

3.3 在什么情况下你希望准确地知道哪一个错误编码是最后报告的？

3.4 修改测试程序 terrno.c，使对所有有效的错误编码调用 perror。<errno.h> 中定义的宏 _NERR 的值比最大的有效错误编码值大 1。

3.5 假设有一个函数 int_Getfcc(void)，它返回 0、EDOM 或者 ERANGE。这些值可以反映函数的上一次调用之后的最后一个浮点错误（如果有的话）。写出一个版本的 <errno.h>，该版本能使用这个函数来搜集程序在使用存储在 errno 中的值的时候出现的浮点错误。

邮
电

3

3.6 [**难**] 编写 <errno.h> 的一个版本，该版本可以将存储在 errno 中的值排队，而且当程序使用这些值时，可以把这些值按顺序返回。什么时候把一个值从队列中移除是安全的？

3.7 [**很难**] 消除 C 标准库中使用 errno 的必要：考虑一下每个可能在 errno 中存储值的函数。保证每个函数都有一种方式来指定几个不同的错误返回值。

第 4 章 <float.h>

4.1 背景知识

浮点算术非常复杂，很多小的处理器在硬件指令方面甚至不支持浮点算术，其他的则需要一个独立的协处理器来处理这种运算，只有最复杂的计算机才会在标准硬件指令集中支持浮点运算。

出于实际的考虑，芯片设计人员经常省略了浮点算术，因为实现浮点比较、加法、减法、乘法和除法花费的微码的数量和实现其他所有指令的相同。通过舍弃浮点支持，本质上可以把微处理器的复杂度减半。

很多应用程序根本不需要浮点算术，其他的可以通过软件来实现浮点运算，只不过需要适度容忍不良的性能，和几千字节的额外代码。只有很少的应用程序需要高性能算术运算，但它们对硬件的要求也很昂贵。所以，一个协处理器的额外花费是可以接受的。

历史

早期的 C 工作在 PDP-11/45 计算机下，那就严重影响了 C 处理浮点算术的方式。例如，*float* 类型（32 位格式）和 *double* 类型（64 位格式）在 C 刚出现的时候就存在了，它们正是 PDP-11 支持的两种格式。但对于这样一个比较小的系统实现语言来说，那就有点不寻常了。

PDP-11/45 FPP 可以在两种模式下工作，它可以对 32 位或者 64 位操作数做所有的算术运算，但必须通过一条指令在两种模式间转换。另一方面，载入并转换一个错误位数的操作数，就像载入一个正确位数的操作数那样简单。这就强烈地鼓励 FPP 只保留一种模式。这样，我们就能解释为什么多年来 C 语言对任何涉及浮点操作数的操作符都产生 *double* 类型的结果，即使两个操作数都是 *float* 类型。就算 FORTRAN 也没有这么慷慨。

当 C 移植到其他的计算机体系下时，这些优点有时候变成了一个麻烦。那些希望支持全部语言的编译器作者不得不为一些非常小的机器编写浮点运算软件，实际上这并不简单。在那些作为标准硬件支持浮点运算的机器上也出现了一系列的问题，可能是因为它们的格式有轻微的不同。这就使编写可移植代码更加困难，你需要编写数学函数和转换算法来使它们的值在一定的范围和精度内变化。

那些提供浮点运算作为选项的机器无疑是最坏的情况，至少对编译器的实现者来说是这样的。首先，那些实现者一定要为那些没有这些选项的机器提供软件支持。当机器有这个选项的时候他们还必须利用那些机器指令。同时有些糊涂的用户无意间链接了两种代码风格或者库中的错误版本，因此编译器作者还要处理这些用户的问题。而且在那些保存中间结果的地方，浮点支持的硬件和软件版本很少情况下会一致。

然而，从语言的立场来说，这些问题中的大部分都是毫无联系的，因此C标准的起草者必须处理的主要问题变化太多了。充分发挥机器的性能是C的优良传统。即使一个右移操作符也要按照底层机器最快的方式执行，更不用说一个浮点加法运算符了。当然，这两种结果都达不到数学家的要求。

上溢和下溢

对浮点算术来说，上溢和下溢是两个突出的问题。一个结果可能太大了而不能在一台机器上表示，但对另一台机器却不是这样。结果上溢可能会导致中断，也可能会生成一个特殊的编码值，或者会产生一个很容易被误认为一个有效值的垃圾值。一个结果也可能太小了而不能在一台机器上表示，但在另一台机器上却可以表示。结果下溢可能会导致中断，也可能偷偷地被一个精确的零代替，这样的0修正（zero fixup）通常是一个不错的选择，但不总是这样。新手总是会写出容易受溢出影响的代码。浮点数所能表示的大范围值引发无知的编程者不注意溢出这一点。其实，我们首先应该做的就是估计结果的量级，避免无谓的误差。

有效值丢失

有时还会遇到更隐晦的问题——有效值丢失。浮点算术允许表示非常多的数值，但这是有代价的。一个值只能表示为固定的精度。把两个精确的数相乘后你可能只能得到一半的有效值。把两个大小相近的数相减，可能会丢失大部分或者全部的有效值。

普通程序员经常会遇到意想不到的有效值丢失。有些课本上看起来非常好的公式转换为代码的时候却漏洞百出。你很难发现在一个序列的临近单元之间变换符号的危险，直到你被它们所困扰。那就是说，你要学会执行形式上而不是实际运行的减法。

上溢、下溢和有效值丢失对浮点算术来说是固有的。它们在一个给定的计算机体系下就很难对付了，编写出具有可移植性的代码更难。起草一个说明怎样写出可移植代码的标准比它还要难。但是，还有一个更糟糕的问题。

变化

两台机器可以使用相同的浮点值表示方法。然而，在每台机器下把两个相同的值相加，可能会得到不同的结果。那些结果在一定程度上依赖于那两台机器的舍入方式。它们可以把那些不能表示的值直接截去并舍入到和它最接近的可表示的值，或者执行一些其他相似的但还有些微小差别的操作。

或者你仅仅可以得到一个错误的结果。在某些场合下，得到一个快速的答案比得到一个尽可能精确的答案要好得多。Seymour Cray 已经成功地建立了几个计算机公司来满足有这种需求的客户，这些机器在某些地方对精度的处理几乎就是保留最低有效位。有时它固定一位或者两位，即使它们有更多的有效位。甚至有些计算机（不是 Cray 设计的）在每次乘以 1 的时候都至少去掉 4 位最低有效位。

如果 C 标准宣布这种行为是非法的，它就永远不会被认可。但有太多的机器使用这种快餐式（quick-and-dirty）的浮点算术计算方法，太多的人仍然使用这些机器。否定了它们，与 C 标准一致的 C 编译器在商业上将是不可接受的。

浮点型的描述

于是，浮点算术领域的 C 标准通常都是描述性的。它尽量定义足够多的术语来讨论浮点型的参变量，但是对怎样才算是得到了一个正确的结果，它几乎没有说明。

X3J11 委员会加入了头文件 `<float.h>` 来补充已经存在的头文件 `<limits.h>`。我们把那些可能对严格的数值运算程序员有用的每一个参数都加入到 `<float.h>` 中。从这些宏中，你就可以充分地了解运行环境的属性。这样，你就可以更好地编写数值算法。（和我前面的立场不同，帮助这类程序员的主要动力来自 Cray Research。）

4.2 C 标准的内容

库部分对 `<float.h>` 的说明非常少。

> **7.1.5 范围 `<float.h>` 和 `<limits.h>`**
>
> 头文件 `<float.h>` 和 `<limits.h>` 定义了几个可以展开为各种范围和参数的宏。
>
> 这些宏，及它们的意思和对它们的值的限定（或者约束）都在 5.2.4.2 中列出来了。

对 `<float.h>` 的详细说明在环境部分。

> **5.2.4.2.2 浮点类型的特征 `<float.h>`**

`<float.h>`

> 浮点类型的特征是在一个模型的基础上定义的，这个模型描述了浮点数字和一些值的表示方法，这些值提供了一个实现的浮点算术的信息[10]。下面的参数被用来定义每一个浮点类型的模型。
>
> s 符号（±1）
>
> b 指数表示的基数（一个大于 1 的整数）
>
> e 指数（一个值在 e_{min} 和 e_{max} 之间的整数）
>
> p 精度（b 进制数的有效位数）
>
> f_k 比 b 小的非负整数（有效数字）
>
> 一个规格化的浮点数 x（如果 $x \neq 0$，则 $f_1 > 0$）由下面的模型定义：

$$x = s \times b^e \times \sum_{k=1}^{p} f_k \times b^{-k} \text{ , } e_{\min} \leqslant e \leqslant e_{\max}$$

头文件 <float.h> 的值中，FLT_RADIX 是一个适合用在 #if 预处理指令中的常量表达式，其他所有的值都不必是常量表达式。除了 FLT_RADIX 和 FLT_ROUNDS，其他所有的值对所有 3 种浮点类型都有不同的名字。除了 FLT_ROUNDS，其他所有的值都由浮点类型模型表示。

FLT_ROUNDS

浮点加法的舍入模式由 FLT_ROUNDS 的值确定其对应关系如下：

-1 不能确定；

0 向 0 舍入；

1 最近舍入；

2 向正无穷大舍入；

3 向负无穷大舍入。

FLT_ROUNDS 的其他所有值都描述为实现定义的舍入行为。

下面列出的值将被实现定义的表达式代替，这些表达式在数值上（绝对值）应该等于或者大于给出的值，符号相同。

FLT_RADIX

- 指数表示的基数 b

```
FLT_RADIX                          2
```

FLT_MANT_DIG
DBL_MANT_DIG
LDBL_MANT_DIG

- FLT_RADIX 进制的浮点数的有效位数 p

```
FLT_MANT_DIG
DBL_MANT_DIG
LDBL_MANT_DIG
```

FLT_DIG
DBL_DIG
LDBL_DIG

- 小数位数 q，满足任何具有 q 个小数位的浮点数都可以舍入到一个具有 p 个有效数字的 b 进制数，而且不用改动就可以回到 q 个小数位，

$$\lfloor (p-1) \times \log_{10} b \rfloor + \begin{cases} 1 \text{ 如果 } b \text{ 是10的幂} \\ 0 \text{ 否则} \quad \text{e} \end{cases}$$

```
FLT_DIG                            6
DBL_DIG                           10
LDBL_DIG                          10
```

FLT_MIN_DIG
DBL_MIN_DIG
LDBL_MIN_DIG

- 最小负整数 $e_{\min}$，满足 FLT_RADIX 的 $(e_{\min}-1)$ 次幂也是一个规格化的浮点数，

```
FLT_MIN_EXP
DBL_MIN_EXP
LDBL_MIN_EXP
```

FLT_MIN_10_EXP
DBL_MIN_10_EXP
LDBL_MIN_10_EXP

- 最小负整数 $\log_{10} b^{e_{\min}}$ ，使 10 的 $\log_{10} b^{e_{\min}}$ 次幂也在规格化浮点数的范围内。

```
FLT_MIN_10_EXP                   -37
DBL_MIN_10_EXP                   -37
LDBL_MIN_10_EXP                  -37
```

FLT_MAX_EXP
DBL_MAX_EXP
LDBL_MAX_EXP

- 最大整数 $e_{\max}$，使 FLT_RADIX 的 $(e_{\max}-1)$ 次幂是一个可表示的有限浮点数。

```
FLT_MAX_EXP
DBL_MAX_EXP
LDBL_MAX_EXP
```

FLT_MAX_10_EXP
DBL_MAX_10_EXP
LDBL_MAX_10_EXP

- 最大整数 $\log_{10}(1-b^{-p})\times b^{e_{max}}$，使 10 的 $\log_{10}(1-b^{-p})\times b^{e_{max}}$ 次幂也在可表示的有限浮点数的范围内。

```
FLT_MAX_10_EXP                    +37
DBL_MAX_10_EXP                    +37
LDBL_MAX_10_EXP                   +37
```

下面列出的值可以被实现定义的表达式代替，这些表达式的值等于或者大于那些给出的值。

FLT_MAX
DBL_MAX
LDBL_MAX

- 最大可表示的有限浮点数$(1-b^{-p})\times b^{e_{max}}$

```
FLT_MAX                          1E+37
DBL_MAX                          1E+37
LDBL_MAX                         1E+37
```

下面列出的值可以被实现定义的表达式代替，这些表达式的值等于或者小于那些给出的值。

FLT_EPSILON
DBL_EPSILON
LDBL_EPSILON

- 1 和给定的浮点类型可表示的比 1 大的最小值之差，b^{1-p}

```
FLT_EPSILON                       1E-5
DBL_EPSILON                       1E-9
LDBL_EPSILON                      1E-9
```

FLT_MIN
DBL_MIN
LDBL_MIN

- 最小的正的规格化浮点数 $b^{e_{min}-1}$

```
FLT_MIN                          1E-37
DBL_MIN                          1E-37
LDBL_MIN                         1E-37
```

例子

下面的例子描述了满足这个国际标准最低需要的人工的浮点数表示和适合头文件 <float.h> 中 float 类型的值。

$$x = s\times 16^{e}\times\sum_{k=1}^{6} f_k\times 16^{-k},\ -31\leqslant e\leqslant +32$$

```
FLT_RADIX                           16
FLT_MNAT_DIG                         6
FLT_EPSILON              9.53674316E-07F
FLT_DIG                              6
FLT_MIN_EXP                        -31
FLT_MIN                  2.93873588E-39F
FLT_MIN_10_EXP                     -38
FLT_MAX_EXP                        +32
FLT_MAX                  3.40282347E+38F
FLT_MAX_10_EXP                     +38
```

下面的例子描述了满足 ANSI/IEEE 754-1985[11] 单精度和双精度规格化数需要的浮点数表示，和适合头文件 <float.h> 中 float 和 double 类型的值。

$$x_f = s\times 2^{e}\times\sum_{k=1}^{24} f_k\times 2^{-k},\ -125\leqslant e\leqslant +128$$

$$x_d = s\times 2^{e}\times\sum_{k=1}^{53} f_k\times 2^{-k},\ -1021\leqslant e\leqslant +1024$$

```
FLT_RADIX                            2
FLT_MANT_DIG                        24
FLT_EPSILON              1.19209290E-07F
FLT_DIG                              6
FLT_MIN_EXP                       -125
FLT_MIN                  1.17549435E-38F
FLT_MIN_10_EXP                     -37
```

```
FLT_MAX_EXP                              +128
FLT_MAX                        3.40282347E+38F
FLT_MAX_10_EXP                            +38
DBL_MANT_DIG                               53
DBL_EPSILON             2.2204460492503131E-16
DBL_DIG                                    15
DBL_MIN_EXP                             -1021
DBL_MIN                2.2250738585072014E-308
DBL_MIN_10_EXP                           -307
DBL_MAX_EXP                             +1024
DBL_MAX                1.7976931348623157E+308
DBL_MAX_10_EXP                           +308
```

参见：条件包含（6.8.1）。

脚注

10. 浮点模型是为了阐述每一个浮点特征的描述，它不要求实现的浮点算术是一致的。

11. 标准中的浮点模型从 *b* 的零次幂进行求和，所以指数范围的值比这里给出的数小 1。

4.3 <float.h> 的使用

只有那些最复杂的数值程序才会关心 <float.h> 中定义的大多数宏，或者适应不同浮点表示之间的变化。我只在某些场合才发现这些参数的用途，你也只会在库中的某些地方看到它们的价值。然而，那有点误导的意思了。在有些地方，我使用了 <float.h> 中的宏用到的底层的宏（见 4.4 节）。在其他的地方，这些代码包含了某些浮点参数的取值范围或者最大值的隐含的假设，但这又限制了可移植性。

使用这些宏可以提前发现某些会造成影响的问题。记住，浮点算术的 3 个缺陷是上溢、下溢和有效值丢失。这里有几种使用 <float.h> 中定义的宏的方式，它可以让你更加安全地执行 *double* 型算术运算。当然，它们也适用于 *float* 和 *long double* 类型。

上溢 为了避免上溢，一定要保证所有的值都不会超过 DBL_MAX 的数值。当然，像下面这样测试最终结果对你没有任何好处：

```
if (DBL_MAX < fabs(y)) /* SILLY TEST */
    ...
```

（这里和下面的例子中出现的函数都是 <math.h> 声明的一般数学函数）

执行测试的时候，错误已经发生了。如果存储在 y 中的值太大了而不能表示，y 可能会包含一个特殊的代码：DBL_MAX 的值或者一个没用的值——这由实现提供的浮点算术的种类所决定。或者可能会在计算这个值的时候终止执行。在任何情况下，上面的测试都不会产生一个有用的结果。一个更有意义的测试是：

```
if (x < log(DBL_MAX))
    y = exp(x);
else
    ...         /* HANDLE OVERFLOW */
```

可以通过使用一个相关的宏来避免计算 log(DBL_MAX)，就像：

```
if (x <= FLT_MAX_10_EXP)
    y = pow(10, x);
else
    ...         /* HANDLE OVERFLOW */
```

如果 FLT_RADIX 不等于 10，这个测试就更有必要了。（现代计算机通常使 FLT_RADIX 的值等于 2，或者，更少的情况下，等于 16。）如果要编写可以接收所有可能的输入的函数，那就要另当别论了。否则，这个测试已经足够严密了。

函数 ldexp 使得通过 2 的幂来换算一个浮点数变得很容易。在 FLT_RADIX 等于 2 的普遍情况下，那是一个效率很高的操作。对一个整型指数 n 来说，可以作以下的简单测试：

```
if (n < FLT_MAX_EXP)
    y = ldexp(1.0, n);
else
    ...         /* HANDLE OVERFLOW */
```

当你为数学库编写附加的函数的时候，很有可能使用这最后一个测试。

下溢 为了避免下溢，一定要保证所有的值都大于等于 DBL_MIN 的数值。下溢的结果不像上溢那样损失惨重，但是它仍然能制造麻烦。IEEE 754 浮点算术提供了*渐进下溢*（gradual underflow），这就减轻了下溢的一些最坏的影响。几乎所有的浮点实现都用零值来代替一个过小而不能表示的值。只有当除以一个会产生下溢的值时，才会遇到麻烦。因为令人想不到的是，你的程序会遭遇到一个零除数，并随之产生一些混乱。可以作以下测试：

```
if (fabs(y) < DBL_MIN)
    ...    /* UNDERFLOW HAS OCCURRED */
```

这并不像相应的跟 DBL_MAX 的比较那样愚蠢，但仍然是在损害发生之后才作检查。当然，也可以作如下相应的测试：

```
if (log(DBL_MIN) <= x)
        y = exp(x);
    else
        ...         /* HANDLE UNDERFLOW */

if (FLT_MIN10_EXP <= x)
        y = pow(10,x);
    else
        .....       /* HANDLE UNDERFLOW */

if (FLT_MIN_EXP < n)
        y = ldexp(1.0, n);
    else
        ...         /* HANDLE UNDERFLOW */
```

有效值丢失 当对两个几乎相等的值相减时就会发生有效值丢失，这种错误没法纠正，除非在编写代码之前仔细地分析这个问题。然而，可以阻止有效值丢失的隐秘形式——把一个很小的数和一个很大的数相加。一个浮点数表示只能保持一个确定的精度。因此，在加的过程中，较小的数的重要作用可能就体现不出来了。

例如，求积分（用一些离散量的总和来近似地表示一个连续的量）的时候，就会陷入困境。一种积分计算方法是通过把曲线下面的一系列矩形的面积加起来计算一个曲线覆盖的面积。很明显，这些矩形越窄，它们的和就越跟曲线覆盖的面积接近。但不幸的是，这只在理论上成立。把一个非常小的矩形面积加到当前的和中，它的部分或者全部作用就无法体现。例如，可以通过下面的代码测试一下，x 和 y 相加是否会使 y 的至少 3 个十进制有效位丢失（假设两者都为正数）：

```
if (x < y* DBL_EPSILON* 1.0E+03)
    ...   /* HANDLE SIGNIFICANCE LOSS */
```

其他的宏 最不可能用到的两个宏是 FLT_RADIX 和 FLT_ROUNDS。事实上，虽然我在这里简单地总结了一下，但如果你从来没有机会使用 <float.h> 中定义的任何宏，也不要感到奇怪。

你应该对浮点算术的特性和缺陷有所了解，也应该知道可移植的 C 代码和为日常使用的机器编写的代码中浮点值的安全范围和精度。你可能会使用 <float.h> 中定义的一些宏来将安全检查固定到你的代码中。但是，不要认为这个头文件包含了编写高度可移植代码的关键因素，因为它并没有。

4.4 <float.h> 的实现

原则上，这个头文件除了一串宏定义，其他什么都没有。对于一个特定的实现，只不过是确定一下参数的值并把它们加进去，甚至可以使用一个叫做 enquire 的免费程序来自动地生成 <float.h>。

对于浮点算术来说，近年来常用的实现都以 IEEE 754 标准为基础。你将发现在 Intel 80X87 和 Motorola MC680X0 协处理器下的 IEEE 754 浮点算术，只有两个常用的文本行命名。它是一个复杂的标准，但是只有它的总体属性影响着 <float.h>。在 IEEE 754 标准下，*long double* 类型可以有一个 80 位的表示，但它经常和 *double* 类型有相同的表示。在这种普遍的情况下，可以考虑把 C 标准的例子中的值复制出来（参考 4.2 节）。

然而，你可能会发现一些问题，并不是所有的翻译器都擅长转换浮点常数。有些会固定最低有效位或者两个有效位。在某些极限值（例如 DBL_MAX 和 DBL_MIN）的情况下，那就可能产生溢出，或者可能会破坏其他值（例如 DBL_EPSILON）的关键行为。

使用联合 所以，我们至少应该检查浮点值产生的位模式。可以通过把这个值以一种方式添加到一个联合中，然后以另一种方式把它提取出来以实现上述的检查，例如：

```
union {
    double _D;
    unsigned short _Us[4];
    } dmax = DBL_MAX;
```

这里假设 *unsigned short* 占据 16 位，*double* 是 IEEE 754 的 64 位表示。有些计算机把最高有效字存储在 dmax._Us[0] 中，其他的存储在 dmax._Us[3] 中。你必须检查你的实现做了什么。不管是哪种情况，最高有效字的值应该等于 0x7FEF，所有其他字的值应该等于 0xFFFF。

一个更加安全的方法和它很相似。把联合初始化为一个位模式的序列，然后定义这个宏并通过联合的一个浮点成员访问该联合。因为只能初始化联合的第一个成员，所以必须把上面例子中的成员声明顺序颠倒过来。用这种方法，可以把下面的代码放入 <float.h>：

```
typedef union {
    unsigned short _Us[4];
    double _D;
    } _Dtype;
extern _Dtype _Dmax, _Dmin, _Deps;
#define DBL_MAX     _Dmax._D;
```

在一个库的源文件中，要提供 _Dmax 和其他相关的定义。对于先存储最低有效字的 80X86 系列，可以编写：

```
#include <float.h>
_Dtype _Dmax = {{0xffff, 0xffff, 0xffff, 0x7fef}};
```

这些代码现在不太容易读懂，但它非常健壮。图 4-1 显示了随之产生的 float.h 版本。每一个宏都能够引用 3 个类型为 _Dvals—_Dbl、_Flt 和 _Ldbl 的数据对象中的一个。一个名为 xfloat.c 的文件定义了这些数据对象。

在编写这些相应的数据对象的时候，我遇到了另一个问题：不同的浮点格式，要求不同的初始化。即使遵循 IEEE 754 标准，不论 *long double* 占据 64 位还是 80 位，都必须指明存储在一个数据对象中的字节的顺序。FLT_RADIX 等于 2 的其他格式只在一些细小的方面有所不同。

参数 现在我们需要再一次确定编码的参数。在 3.4 节，我介绍了内部头文件 <yvals.h>，把那些随翻译器变化的参数放在该文件中。错误编码是这样的参数的一个集合，浮点表示的属性构成了另一个集合。如果一个库的源文件在 C 的不同实现中必须在某些细小的方面作些改变，那么就可以在这些库的源文件中包含 <yvals.h>。

图 4-1
float.h

```
/* float.h standard header -- IEEE 754 version */
#ifndef _FLOAT
#define _FLOAT
#ifndef _YVALS
#include <yvals.h>
#endif
        /* type definitions */
typedef struct {
    int _Ddig, _Dmdig, _Dmax10e, _Dmaxe, _Dmin10e, _Dmine;
    union {
        unsigned short _Us[5];
        float _F;
        double _D;
        long double _Ld;
        } _Deps, _Dmax, _Dmin;
    } _Dvals;
        /* declarations */
extern _Dvals _Dbl, _Flt, _Ldbl;
        /* double properties */
#define DBL_DIG         _Dbl._Ddig
#define DBL_EPSILON     _Dbl._Deps._D
#define DBL_MANT_DIG    _Dbl._Dmdig
#define DBL_MAX         _Dbl._Dmax._D
#define DBL_MAX_10_EXP  _Dbl._Dmax10e
#define DBL_MAX_EXP     _Dbl._Dmaxe
#define DBL_MIN         _Dbl._Dmin._D
#define DBL_MIN_10_EXP  _Dbl._Dmin10e
#define DBL_MIN_EXP     _Dbl._Dmine
       /* float properties */
#define FLT_DIG         _Flt._Ddig
#define FLT_EPSILON     _Flt._Deps._F
#define FLT_MANT_DIG    _Flt._Dmdig
#define FLT_MAX         _Flt._Dmax._F
#define FLT_MAX_10_EXP  _Flt._Dmax10e
#define FLT_MAX_EXP     _Flt._Dmaxe
#define FLT_MIN         _Flt._Dmin._F
#define FLT_MIN_10_EXP  _Flt._Dmin10e
#define FLT_MIN_EXP     _Flt._Dmine
       /* common properties */
#define FLT_RADIX       2
#define FLT_ROUNDS      _FRND
       /* long double properties */
#define LDBL_DIG        _Labl._Ddig
#define LDBL_EPSILON    _Ldbl._Deps._Ld
#define LDBL_MANT_DIG   _Ldbl._Dmdig
#define LDBL_MAX        _Ldbl._Dmax._Ld
#define LDBL_MAX_10_EXP_Ldbl._Dmax10e
#define LDBL_MAX_EXP    _Ldbl._Dmaxe
#define LDBL_MIN        _Ldbl._Dmin._Ld
#define LDBL_MIN_10_EXP_Ldbl._Dmin10e
#define LDBL_MIN_EXP    _Ldbl._Dmine
#endif
```
□

<yvals.h> 定义了下面的参数：

_DO

- _DO 是用来表示 *double* 值的由 4 个 *unsigned short* 类型组成的数组的最高有效元素的下标。它的值不是 0 就是 3。（具有其他 3 个下标（_D1，_D2 和 _D3）的宏在 _D0 的基础上定义，它们在库的其他地方会被用到。）

_DOFF
_FOFF
_LOFF

- _DOFF 是最高有效小数元素中的小数位 FFF... 的个数。该元素的最高有效位是浮点值的符号 s，值为 0 或者 1。其他的位作为一个无符号位域表示特征值 ccc...。参考图 4-2 *double* 类型表示的格式。_FOFF 是 *float* 类型的相应值。_LOFF 是 *long double* 类型的相应值。

_DBIAS
_FBIAS
_LBIAS

- _DBIAS 是一个从 *double* 的特征值中减掉的以确定它的指数的值。_FBIAS 是 *float* 类型的相应的值。_LBIAS 是 *long double* 类型的相应的值。小数值 F 是 1.FFF...（*float* 和 *double* 型）或者 0.FFF...（IEEE 754 80 位格式的 *long double* 型），在这里 FFF... 是小数位。那么，一个 *double* 数的值为：

$$-1^{s} * (1.FFF...) * 2^{(ccc...)-_DBIAS}$$

_DLONG

- 如果 *long double* 具有 IEEE 754 的 80 位格式，那么 _DLONG 为非零。

_FRND

- _FRND 是宏 FLT_ROUNDS 的值。

xfloat.c

图 4-3 显示了 xfloat.c 的代码，它是在这些参数的基础上编写的。这段代码也包含了一些隐含的假设：

- FLT_RADIX 的值为 2。
- *float* 类型为 32 位表示，并且一定和含有两个 *unsigned short* 类型的数组重叠，而 *double* 类型为 64 位表示，一定与由 4 个 *unsigned short* 类型的数组重叠。
- 只有 _DLONG 为非零时，*long double* 类型才具有 IEEE 754 的 80 位表示；否则，它和 *double* 的表示相同。
- 特征值的位数绝不会比 14 更大。
- *float* 或者 *double* 中的小数值中包含一个隐藏位，该位就是上面提到的 FFF... 之前的那个 1。

作为一个例子，下面给出 Intel 80X87 协处理器的相关值，这里假设 *double* 和 *long double* 具有不同的表示：

```
#define _D0      3
#define _DBIAS   0x3fe
#define _DLONG   1
#define _DOFF    4
#define _FBIAS   0x7e
#define _FOFF    7
#define _FRND    1
#define _LBIAS   0x3ffe
#define _LOFF    15
```

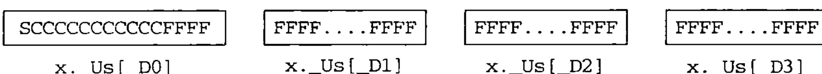


图 4-2 *double* 格式

图 4-3 xfloat.c

```
/* values used by <float.h> macros -- IEEE 754 version */
#include <float.h>

        /* macros */
#define DFRAC (49+_DOFF)
#define DMAXE ((1U<<(15-_DOFF))-1)
#define FFRAC (17+_FOFF)
#define FMAXE ((1U<<(15-_FOFF))-1)
#define LFRAC (49+_LOFF)
#define LMAXE 0x7fff
#define LOG2  0.30103
#if_D0 != 0                                   /* low to high words */
#define DINIT(w0, wx) wx, wx, wx, w0
#define FINIT(w0, wx) wx, w0
#define LINIT(w0, w1, wx) wx, wx, wx, w1, w0
#else                                         /* high to low words */
#define DINIT(w0, wx) w0, wx, wx, wx
#define FINIT(w0, wx) w0, wx
#define LINIT(w0, w1, wx) w0, w1. wx, wx, wx
#endif
        /* static data */
_Dvals _Dbl = {
    (int)((DFRAC-1)*LOG2),                              /* DBL_DIG */
    (int )DFRAC,                                   /* DBL_MANT_DIG */
    (int)((DMAXE-_DBIAS-1)*LOG2),                /* DBL_MAX_10_EXP */
    (int)(DMAXE-_DBIAS-1),                          /* DBL_MAX_EXP */
    (int)(-_DBIAS*LOG2),                         /* DBL_MIN_10_EXP */
    (int)(1-_DBIAS),                                /* DBL_MIN_EXP */
    {DINIT(_DBIAS-DFRAC+2<<_DOFF, 0)},              /* DBL_EPSILON */
    {DINIT((DMAXE<<_DOFF)-1, ~0)},                      /* DBL_MAX */
    {DINIT(1<<_DOFF, 0)},                               /* DBL_MIN */
    };
_Dvals _Flt ={
    (int)((FFRAC-1)*LOG2),                              /* FLT_DIG */
    (int)FFRAC,                                    /* FLT_MANT_DIG */
    (int)((FMAXE-_FBIAS-1)*LOG2),                /* FLT_MAX_10_EXP */
    (int)(FMAXE-_FBIAS-1),                          /* FLT_MAX_EXP */
    (int)(-_FBIAS*LOG2),                         /* FLT_MIN_10_EXP */
    (int)(1-_FBIAS),                                /* FLT_MIN_EXP */
    {FINIT(_FBIAS-FFRAC+2<<_FOFF, 0)},              /* FLT_EPSILON */
    {FINIT((FMAXE<<_FOFF)-1,~0)},                       /* FLT_MAX */
    {FINIT(1<<_FOFF, 0)},                               /* FLT_MIN */
    };
#if _DLONG
_Dvals _Ldbl = {
    (int)((LFRAC-1)*LOG2),                             /* LDBL_DIG */
    (int)LFRAC,                                   /* LDBL_MANT_DIG */
    (int)((LMAXE-_LBIAS-1)*LOG2),               /* LDBL_MAX_10_EXP */
    (int)(LMAXE-_LBIAS-1),                         /* LDBL_MAX_EXP */
    (int)(-_LBIAS*LOG2),                        /* LDBL_MIN_10_EXP */
    (int)(1-_LBIAS),                               /* LDBL_MIN_EXP */
    {LINIT(_LBIAS-LFRAC+2, 0x8000, 0)},            /* LDBL_EPSILON */
    {LINIT(LMAXE-1, ~0, ~0)},                          /* LDBL_MAX */
    {LINIT(1, 0x8000, 0)},                             /* LDBL_MIN */
    };
```

图 4-3
（续）

```
#else
_Dvals _Lab1 = {
    (int)(DFRAC*LOG2),                          /* LDBL_DIG */
    (int)DFRAC,                            /* LDBL_MANT_DIG */
    (int)((DMAXE-_DBIAS-1)*LOG2),        /* LDBL_MAX_10_EXP */
    (int)(DMAXE-_DBIAS-1),                  /* LDBL_MAX_EXP */
    (int) (-_DBIAS*LOG2),                /* LDBL_MIN_10_EXP */
    (int) (1-_DBIAS),                       /* LDBL_MIN_EXP */
    {DINIT(_DBIAS-DFRAC+2<<_DOFF,0)},       /* LDBL_EPSILON */
    {DINIT((DMAXE<<_DOFF)-1,~0)},               /* LDBL_MAX */
    {DINIT(1<<_DOFF, 0)},                       /* LDBL_MIN */
    };
#endif
```

□

4.5 <float.h> 的测试

图 4-4 显示了测试程序 tfloat.c。程序一开始就输出 <float.h> 中定义的宏的值，而且输出方式便于理解。然后，它检查这些宏是否满足 C 标准描述的最低要求。

下面是 Intel 80X87 协处理器在一个实现下的输出，该实现支持所有 3 种大小的 IEEE 754 操作数。

```
FLT_RADIX = 2

DBL_DIG =             15     DBL_MANT_DIG =      53
DBL_MAX_10_EXP =     308     DBL_MAX_EXP =     1024
DBL_MIN_10_EXP =    -307     DBL_MIN_EXP =    -1021
      DBL_EPSILON =    2.220446e-16
      DBL_MAX =        1.797693e+308
      DBL_MIN =        2.225074e-308

FLT_DIG =              6     FLT_MANT_DIG =      24
FLT_NAX_10_EXP =      38     FLT_MAX_EXP =      128
FLT_MIN_10_EXP =     -37     FLT_MIN_EXP =     -125
      FLT_EPSILON =    1.192093e-07
      FLT_MAX =        3.402823e+38
      FLT_MIN =        1.175494e-38

LDBL_DIG =            19     LDBL_MANT_DIG =     64
LDBL_MAX_10_EXP =   4932     LDBL_MAX_EXP =   16384
LDBL_MIN_10_EXP =  -4931     LDBL_MIN_EXP =  -16381
      LDBL_EPSILON = 1.084202e-19
      LDBL_MAX =     1.189731e+4932
      LDBL_MIN =     3.362103e-4932
SUCCESS testing <float.h>
```

在开发 <float.h> 和 xfloat.c 的过程中，我没有发现任何错误。大部分错误都是通过运行 tfloat.c 才发现的。因而这些测试其实并不简单。

图 4-4
tfloat.c

```
/* test float macros */
#include <assert.h>
#include <float.h>
#include <math.h>
#include <stdio.h>

int main()
    {                       /* test basic properties of float.h macros */
    double radlog;
    int digs;
    static int radix = FLT_RADIX;

    printf("FLT_RADIX = %i\n\n",FLT_RADIX);
    printf("DBL_DIG =        %5i  DBL_MANT_DIG = %6i\n",
        DBL_DIG, DBL_MANT_DIG);
    printf("DBL_MAX_10_EXP = %5i  DBL_MAX_EXP = %6i\n",
        DBL_MAX_10_EXP, DBL_MAX_EXP);
    printf("DBL_MIN_10_EXP = %5i  DBL_MIN_EXP = %6i\n",
        DBL_MIN_10_EXP, DBL_MIN_EXP);
    printf("       DBL_EPSILON = %le\n", DBL_EPSILON);
    printf("       DBL_MAX =     %le\n", DBL_MAX);
    printf("       DBL_MIN =     %le\n\n", DBL_MIN);
    printf("FLT_DIG =          %5i FLT_MANT_DIG = %6i\n",
        FLT_DIG, FLT_MANT_DIG);
    printf("FLT_MAX_10_EXP =   %5i FLT_MAX_EXP = %6i\n",
        FLT_MAX_10_EXP, FLT_MAX_EXP);
    printf("FLT_MIN_10_EXP =   %5i FLT_MIN_EXP = %6i\n",
       FLT_MIN_10_EXP, FLT_MIN_EXP);
    printf("     FLT_EPSILON =  %e\n", FLT_EPSILON);
    printf("     FLT_MAX =      %e\n", FLT_MAX);
    printf("     FLT_MIN =      %e\n\n", FLT_MIN);
    printf("LDBL_DIG=          %5i LDBL_MANT_DIG = %6i\n",
       LDBL_DIG, LDBL_MANT_DIG);
    printf("LDBL_MAX_10_EXP = %5i LDBL_MAX_EXP = %6i\n",
       LDBL_MAX_10_EXP, LDBL_MAX_EXP);
    printf("LDBL_MIN_10_EXP = %5i LDBL_MIN_EXP = %6i\n",
       LDBL_MIN_10_EXP, LDBL_MIN_EXP);
    printf("     LDBL_EPSILON = %Le\n", LDBL_EPSILON);
    printf("     LDBL_MAX =     %Le\n", LDBL_MAX);
    printf("     LDBL_MIN =     %Le\n", LDBL_MIN);
    radlog = log10(radix);
        /* test double properties */
    assert(10 <= DBL_DIG && FLT_DIG <= DBL_DIG);
    assert(DBL_EPSILON <= 1e-9);
    digs = (DBL_MANT_DIG - 1) * radlog;
    assert(digs <= DBL_DIG && DBL_DIG <= digs + 1);
    assert(1e37 <= DBL_MAX);
    assert(37 <= DBL_MAX_10_EXP);
#if FLT_RADIX == 2
    assert(ldexp(1.0, DBL_MAX_EXP - 1) < DBL_MAX);
    assert(ldexp(1.0, DBL_MIN_EXP _ 1) == DBL_MIN);
#endif
    assert(DBL_MIN <= 1e-37);
    assert(DBL_MIN_10_EXR <= -37);
```

图 4-4（续）

```
        /* test float properties */
    assert(6 <= FLT_DIG);
    assert(FLT_EPSILON <= 1e-5);
    digs = (FLT_MANT_DIG - 1) * radlog;
    assert(digs <= FLT_DIG && FLT_DIG <= digs + 1);
    assert(1e37 <= FLT_MAX);
    assert(37 <= FLT_MAX_10_EXP);
#if FLT_RADIX == 2
    assert(ldexp(1.0, FLT_MAX_EXP - 1) < FLT_MAX);
    assert(ldexp(1.0, FLT_MIN_EXP - 1) == FLT_MIN);
#endif
    assert(FLT_MIN <= 1e-37);
    assert(FLT_MIN_10_EXP <= -37);
        /* test universal properties */
#if FLT_RADIX < 2
#error bad FLT_RADIX
#endif
    assert(-1 <= FLT_ROUNDS && FLT_ROUNDS <= 3);
        /* test long double properties */
    assert(10 <= LDBL_DIG && DBL_DIG <= LDBL_DIG);
    assert(LDBL_EPSILON <= 1e-9);
    digs = (LDBL_MANT_DIG - 1) * radlog;
    assert(digs <= LDBL_DIG && LDBL_DIG <= digs + 1);
    assert(1e37 <= LDBL_MAX);
    assert(37 <= LDBL_MAX_10_EXP);
#if FLT_RADIX == 2
    assert(DBL_MAX_EXP < LDBL_MAX_EXP
        || ldexp(1.0, LDBL_MAX_EXP - 1) < LDBL_MAX);
    assert(LDBL_MIN_EXP < DBL_MIN_EXP
        || ldexp(1.0, LDBL_MIN_EXP - 1) == LDBL_MIN);
#endif
    assert(LDBL_MIN <= 1e-37);
    assert(LDBL_MIN_10_Exp <= -37);
    puts("SUCCESS testing <float.h>");
    return(0);
    }
```

4.6 参考文献

ANSI/IEEE Standard 754-1985 (Piscataway, N.J.: Institute of Electrical and Electronics Engineers, Inc., 1985). 这是在现代微处理器中广泛使用的浮点运算标准。

Jack J. Dongarra and Eric Grosse, "Distribution of Mathematical Software via Electronic Mail," *Communications of the ACM*, 30 (1987), pp. 403-407. 这篇文章告诉你怎样通过电子邮件得到各种测试程序。下面两个可以通过电子邮件获得的程序在浮点算术方面经过了千锤百炼。

- 程序 enquire 用来测试伴随 C 实现的浮点算术的属性。它以一个可用的 float.h 文件的形式打印出测试结果。该程序由 CWI, Amsterdam 的 Steven Pemberton 编写。你可以通过给 steven@cwi.nl 发邮件来获得 enquire。

- 程序 paranoia 可以测试出浮点运算中的很多错误。它最早由加利福尼亚大学伯克利分校（the University of California at Berkeley）的 W.M.Kahan 编写。现在也有了 C 版本。你可以向 netlib@research.att.com 发出以下请求：

  ```
  send paranoia.c from paranoia
  ```

Pat Sterbenz, *Floating-Point Computation* (Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1973). 这本书很老了而且现在已经不出版了。然而，关于这些基本问题的讨论，很难找到比这更好的书了。

4.7 习题

4.1 为你使用的 C 翻译器确定描述浮点运算的参数，它们和 IEEE 754 标准一致吗？

4.2 你可以修改 <yvals.h> 以使它适应你使用的 C 翻译器上的 <float.h> 和 xfloat.c 吗？如果可以，就去试一下；如果不能，还必须修改其他的什么东西吗？

4.3 考虑下面的代码：

```
double d = 1.0;
float a [N];

for(i = 0; i < n; ++i)
    d *= a [i] ;
```

在 IEEE 754 浮点算术中，在 d 的计算溢出之前，N 最大是多少？

4.4 考虑下面的代码：

```
long double ld = 1.0;
double a [N];

for(i = 0; i < n; ++i)
    ld *= a[i];
```

在 IEEE 754 浮点算术中，在 ld 的计算结果溢出之前，N 最大是多少？

4.5 为什么 <yvals.h> 直接包含在 <float.h> 中（而不是只包含在 xfloat.c 中）？修改本章中的代码来消除这种需要。

4.6 给定函数 int_Getrnd(void)，它可以返回当前的浮点舍入状态。修改宏 FLT_RADIX 来返回当前状态。

4.7 [**难**] 编写一个 C 程序，使它仅通过使用算术运算就能确定 <float.h> 定义的所有宏的值。假设你不知道底层的浮点数表示。

4.8 [**很难**] 修改上题中的程序，使它可以在某个实现下安全地工作，该实现遇到浮点数溢出时会异常终止程序。假设一旦发生溢出，程序就不能重新获得控制权。

第5章 <limits.h>

5.1 背景知识

对C程序设计语言的所有部分进行标准化的第一次尝试开始于1980年，它是由一个当时称作/usr/group，现在叫UniForum的组织发起的。作为第一个发展UNIX的商业性的组织，/usr/group对厂商独立标准的形成有着重大影响。这个组织感觉到了技术的发展不能简单地向各个方向进行，他们也不能只单一地为AT&T服务，因为这两种方式都很难维持一个开放性的市场。

历史

所以/usr/group开始了定义UNIX和类UNIX系统的具体含义的过程。他们成立了一个标准委员会，这个委员会至少在刚开始的时候专注于C编程环境。总之，几乎所有的应用程序都在该环境下实现。他们的目标是描述很多C函数，这些函数可以在任何UNIX可兼容的系统中找到。当然，这些描述必须独立于任何特定的体系结构。

/usr/group的描述中一部分是允许访问UNIX系统服务的、C可调用的函数集。不过，更大一部分是对所有的C环境都适用的函数集，这一部分是C标准中库的基础。因为Kernighan和Ritchie选择不去专门讨论库，只是必要时顺便提一下，所以/usr/group标准极大地帮助了X3J11委员会，为我们节省了很多个月甚至几年的额外劳动。

另外，/usr/group的成果还有一个很有用的目的。IEEE委员会1003的成立是为了把这个工业产品转变成一个官方的标准。IEEE小组把系统独立函数部分的责任转交给了X3J11，而 自己则专注于UNIX特定的部分。因而，就有了今天的IEEE 1003.1标准，也叫做POSIX。

为变化的东西命名

要构建一个体系结构独立的描述，一部分工作就是找出不同的计算机体系结构之间有哪些发生了变化。固然，我们希望避免一切不必要的变化，然后剩下的就是要识别和限制的部分了。当一个应用程序移到另一种类型的UNIX下的时候，某个关键的值可能会改变。所以，要对该值进行命名。要制定程序中测试该值的规则。而且要定义该值的变化范围。

C中的一个优良传统是标量数据类型要以一种对每个计算机体系结构都很自然的方式表示。基本类型*int*的弹性很大，它希望自己的大小至少在一

个很宽的范围内能支持高效的计算。这对效率来说可能是个不错的选择，但针对可移植性来说，就真的是个麻烦了。

/usr/group 发明了标准头文件 <limits.h> 来捕捉很多在计算机体系结构之间变化的重要特性。碰巧，这个头文件又专门地处理整数值的变化范围。当 X3J11 决定为浮点类型加入一些相似的数据的时候，我们通过投票决定不覆盖 <limits.h> 中已经存在的内容，而是添加了标准头文件 <float.h>。也许我们也应该把这个已经存在的头文件改名为 <integer.h>，但是我们没有这样做。因为历史的连贯性战胜了格式的整齐。

5.2 C 标准的内容

5.2.4.2 数字范围

一个符合标准的实现应该记录本条目指定的所有的范围，它们将在头文件 <limits.h> 和 <float.h> 中详细说明。

<limits.h>

5.2.4.2.1 整值类型的大小 <limits.h>

下面给出的值会被适合在 #if 预处理指令中使用的常量表达式代替。而且，除了 CHAR_BIT 和 MB_LEN_MAX 之外，后面的都会被类型与根据整值提升而转换的具有相应类型的对象类型相同的表达式替换。它们的实现定义的值在数值上（绝对值）会大于等于那些给出的值，符号相同。

CHAR_BIT
- 除位域之外的最小对象的位数（字节）
  CHAR_BIT 8

SCHAR_MIN
- signed char 类型的对象的最小值
  SCHAR_MIN -127

SCHAR_MAX
- signed char 类型的对象的最大值
  SCHAR_MAX +127

UCHAR_MAX
- unsigned char 类型的对象的最大值
  UCHAR_MAX 255

CHAR_MIN
- char 类型的对象的最小值
  CHAR_MIN “见下面”

CHAR_MAX
- char 类型的对象的最大值
  CHAR_MAX “见下面”

MB_LEN_MAX
- 任何支持区域设置的多字节字符的最大字节数
  MB_LEN_MAX 1

SHRT_MIN
- short int 类型的对象的最小值
  SHRT_MIN -32767

SHRT_MAX
- short int 类型的对象的最大值
  SHRT_MAX +32767

USHRT_MAX
- unsigned short int 类型的对象的最大值
  USHRT_MAX 65535

INT_MIN
- int 类型的对象的最小值
  INT_MIN -32767

INT_MAX
- int 类型的对象的最大值
  INT_MAX +32767

UINT_MAX

- unsigned int 类型的对象的最大值

  UINT_MAX 65535

LONG_MIN

- long int 类型的对象的最小值

  LONG_MIN -2147483647

LONG_MAX

- long int 类型的对象的最大值

  LONG_MAX +2147483647

ULONG_MAX

- unsigned long int 类型的对象的最大值

  ULONG_MAX 4294967295

如果一个 char 类型的对象在一个表达式中被作为一个有符号整数对待，那么 CHAR_MIN 的值就和 SCHAR_MIN 的相同，CHAR_MAX 的值也会和 SCHAR_MAX 的相同。否则，CHAR_MIN 的值就为 0，CHAR_MAX 的值就和 UCHAR_MAX 的相同[9]。

脚注

9. 参考 6.1.2.5。

5.3 <limits.h> 的使用

可以通过两种方式来使用 <limits.h>。其中简单点的方法可以保证你不会编写出一个可笑的程序。例如，假设要表示某个值在 VAL_MIN 和 VAL_MAX 之间的有符号数据，就可以通过写以下代码来防止程序出现翻译错误：

```
#include <assert.h>
#include <limits.h>
#if VAL_MIN < INT_MIN || INT_MAX < VAL_MAX
#error values out of range
#endif
```

然后就可以安全地把数据存储在声明为 *int* 型的数据对象中。

类型适应

使用 <limits.h> 的一种更加复杂的方法是控制一个程序中的类型选择。可以把上面的例子改为：

```
#include <assert.h>
#include <limits.h>
#if VAL_MIN < INT_MIN || INT_MAX < VAL_MAX
    typedef long Val_t;
#else
    typedef int Val_t;
#endif
```

然后就可以把所有在这个范围内变化的数据对象声明为 Val_t 类型。程序就会选择效率更高的类型。

<limits.h> 的出现也是为了废除一种老的、可移植性非常差的编程方法。一些程序尝试通过编写 #if 预处理指令来测试执行环境的属性：

```
#if (-1 + 0x0) >> 1 > 0x7fff
/* must have ints greater than 16 bits */
...
#endif
```

图 5-1
limits.h

```
/* limits.h standard header -- 8-bit version */
#ifndef _LIMITS
#define _LIMITS
#ifndef _YVALS
#include <yvals.h>
#endif
        /* char properties */
#define CHAR_BIT        8
#if _CSIGN
#define CHAR_MAX        127
#define CHAR_MIN        (-127-_C2)
#else
#define CHAR_MAX        255
#define CHAR_MIN        0
#endif
        /* int properties */
#if _ILONG
#define INT_MAX         2147483647
#define INT_MIN         (-2147483647-_C2)
#define UINT_MAX        4294967295
#else
#define INT_MAX         32767
#define INT_MIN         (-32767-_C2)
#define UINT_MAX        65535
#endif
        /* long properties */
#define LONG_MAX        2147483647
#define LONG_MIN        (-2147483647-_C2)
        /* multibyte properties */
#define MB_LEN_MAX _MBMAX
        /* signed char properties */
#define SCHAR_MAX       127
#define SCHAR_MIN       (-127-_C2)
        /* short properties */
#define SHRT_MAX        32767
#define SHRT_MIN        (-32767-_C2)
        /* unsigned properties */
#define UCHAR_MAX       255
#define ULONG_MAX       4294967295
#define USHRT_MAX       65535
#endif
```
□

这些代码认为无论预处理器执行什么样的运算，都和执行环境的情况相同。那些熟练使用交叉编译器的人知道翻译环境可能会和执行环境有显著的不同。为了能使用这样的方法正确地工作，C 标准需要要求翻译器非常相似地模仿执行环境，并且那些共用一个前端的翻译器家族必须调整翻译时的算法来适应每个执行环境。

X3J11 花了很长时间来讨论这些要求。最后，我们认为预处理器的发明不是为了承担这样严格的要求。固然，翻译器必须在很多方面模仿执行环境。例如，它必须在静态存储空间中计算常量表达式，并且至少达到和执行环境一样大的取值范围和精度。但是它也可以大量地定义它自己的环境来实现 `#if` 预处理指令中的运算。

所以在检测执行环境的时候，不能用预处理器做试验。这时必须包含 <limits.h>，并且测试它提供的宏的值。

X3J11 添加到头文件 <limits.h> 中的一个新的宏是 MB_LEN_MAX。可以使用它来为多字节字符分配空间。我将在第 13 章中把这个宏跟多字节函数结合起来讨论。

5.4 <limits.h> 的实现

必须为这个头文件提供的唯一的代码就是该头文件本身。<limits.h> 中定义的所有的宏都可以使用 #if 预处理指令测试，而且它们在实际执行中改动的可能性不大。（但 <float.h> 中定义的宏却不是这样。）

一般的选择 大多数现代计算机都使用 8 位 *char* 类型、两字节 *short* 类型和 4 字节 *long* 类型。

下面是基于这种思想的几个常用的变种：

- *int* 类型是 2 个或者 4 个字节。
- *char* 类型与 *signed char* 和 *unsigned char* 类型的取值范围相同。
- 有符号的值经常使用 2 的补码进行编码，这种编码只有一种形式的零和一个没有对应正值的负值。一个不太常用的是 1 的补码和有符号数值。它们都有两种形式的零（正零和负零），但没有负值。
- 一个多字节字符的字节数可以是任何比零大的值。

因此，我发现写出一个可以扩展为这些常用的选择的 <limits.h> 版本会很方便。图 5-1 显示了文件 limits.h。它包含了 3.4 节介绍的配置文件 <yvals.h>。那个文件也为头文件 <float.h> 提供了一些参数，这些在 4.4 节有描述。另外，<yvals.h> 定义了以下的宏：

_ILONG **_CSIGN** **_C2** **_MBMAX**

- _ILONG——如果 *int* 类型为 4 个字节则它为非零；
- _CSIGN——如果 *char* 类型带符号则它为非零；
- _C2——如果使用 2 的补码进行编码，则为 1，否则为 0；
- _MBMAX——单个多字节字符的在最坏情况下的长度。

宏 _C2 的使用使一个很重要的细节变得很模糊。在使用 2 的补码的机器中，不能简单地赋予 INT_MIN 一个明显的值。例如，在一个 16 位的机器上，字符序列 -32 768 被分解为两个符号，一个负号和一个值为 32 768 的整型常量。后者为 *long* 类型，因为它太大了而不能用 *int* 类型来表示。对这个值取负不会改变它的类型。然而，C 标准要求 INT_MIN 的类型为 *int*。否则，你就可能会对一个看起来没有错误的语句出现的行为感到惊奇，例如：

图 5-2
tlimits.c

```
/* test limits macros */
#include <limits.h>
#include <stdio.h>

int main()
    {                   /* test basic properties of limits.h macros */
    printf("CHAR_BIT = %2i  MB_LEN_MAX = %2i\n\n",
        CHAR_BIT, MB_LEN_MAX);
    printf(" CHAR_MAX = %10i  CHAR_MIN = %10i\n",
        CHAR_MAX, CHAR_MIN);
    printf(" SCHAR_MAX = %10i SCHAR_MIN = %10i\n",
        SCHAR_MAX, SCHAR_MIN);
    printf("UCHAR_MAX = %10u\n\n", UCHAR_MAX);
    printf(" SHRT_MAX = %10i    SHRT_MIN = %10i\n",
        SHRT_MAX, SHRT_MIN);
    printf("USHRT_MAX = %10u\n\n", USHRT_MAX);
    printf("  INT_MAX = %10i    INT_MIN = %10i\n",
        INT_MAX, INT_MIN);
    printf(" UINT_MAX = %10u\n\n", UINT_MAX);
    printf(" LONG_MAX = %10li  LONG_MIN = %10li\n",
        LONG_MAX, LONG_MIN);
    printf ("ULONG_MAX = %10lu\n", ULONG_MAX);
#if CHAR_BIT < 8 || CHAR_MAX < 127 || 0 < CHAR_MIN \
    || CHAR_MAX != SCHAR_MAX && CHAR_MAX != UCHAR_MAX
#error bad char properties
#endif
#if INT_MAX < 32767 || -32767 < INT_MIN || INT_MAX < SHRT_MAX
#error bad int properties
#endif
#if LONG_MAX < 2147483647 || -2147483647 < LONG_MIN \
    || LONG_MAX < INT_MAX
#error bad long properties
#endif
#if MB_LEN_MAX < 1
#error bad MB_LEN_MAX
#endif
#if SCHAR_MAX < 127 || -127 < SCHAR_MIN
#error bad signed char properties
#endif
#if SHRT_MAX < 32767 || -32767 < SHRT_MIN \
    || SHRT_MAX < SCHAR_MAX
#error bad short properties
#endif
#if UCHAR_MAX < 255 || UCHAR_MAX / 2 < SCHAR_MAX
#error bad unsigned char properties
#endif
#if UINT_MAX < 65535 || UINT_MAX / 2 < INT_MAX \
    || UINT_MAX < USHRT_MAX
#error bad unsigned int properties
#endif
#if ULONG_MAX < 4294967295 || ULONG_MAX / 2 < LONG_MAX \
    || ULONG_MAX < UINT_MAX
```

图 5-2（续）

```
#error bad unsigned long properties
#endif
#if USHRT_MAX < 65535 || USHRT_MAX / 2 < SHRT_MAX \
    || USHRT_MAX < UCHAR_MAX
#error bad unsigned short properties
#endif
    puts("SUCCESS testing <limits.h>");
    return(0);
    }
```

```
printf("range is from %d to %d\n", INT_MIN, INT_MAX);
```

唯一安全的途径是使用一个形如（-32 767-1）的表达式来得到这个值。给定对 <limits.h> 进行参数化的方式，你可以免费地学到这个窍门。

还有一个细节也不容忽视。我在前面已经提到，预处理的运算不必模仿执行环境的做法。原则上，你可以在一个 32 位 *long* 类型的主机上为一个 36 位 *long* 类型的执行环境编译程序。然而，这个主机必须强制获得 <limits.h> 中的正确值。那就是说，它必须以至少 36 位 *long* 类型来执行预处理运算。毕竟，X3J11 为实现者所做的说明的范围没有这么广。

5.5 <limits.h> 的测试

图 5-2 显示了测试程序 tlimits.c。它为提供了一个简洁的、健全的、可以在 <limits.h> 下运行的检测。虽然并不是很详尽，但它确实可以辨别这个头文件是不是基本上健全。它也提供了一个可读的摘要，该摘要记录了 <limits.h> 中定义的宏的值。

注意所有的动作都在翻译时发生，因为所有的宏在 #if 预处理指令下都会起作用。如果这个测试编译通过了，它就一定可以运行，并打印出摘要和成功执行的信息，然后在成功的状态下退出。

下面是在一个兼容 PC 的实现下的输出，该实现下，*char* 和 *signed char* 的表示相同。

```
CHAR_BIT = 8 MB_LEN_MAX = 8

 CHAR_MAX =        127    CHAR_MIN =          -128
SCHAR_MAX =        127   SCHAR_MIN =          -128
UCHAR_MAX =        255

 SHRT_MAX =      32767    SHRT_MIN =        -32768
USHRT_MAX =      65535

  INT_MAX =      32767     INT_MIN =        -32768
 UINT_MAX =      65535

 LONG_MAX = 2147483647    LONG_MIN =   -2147483648
ULONG_MAX = 4294967295
SUCCESS testing <limits.h>
```

5.6 参考文献

4.5 节描述的程序 enquire 也生成了文件 limits.h。

IEEE Standard 1003-1987 (Piscataway, N.J.: Institute of Electrical and Electronics Engineers, Inc., 1985). 这是为了让 C 程序可以在 UNIX 和 UNIX 可兼容的操作系统下运行而制定的 POSIX 标准。头文件 <limits.h> 就是因此而产生的。

5.7 习题

5.1 确定你的翻译器中描述整数算术所使用的参数。

5.2 修改 <limits.h>，使它适用于你使用的翻译器。

5.3 阅读下面的代码：

```
int in = 1.0;
short a[N];

for (i = 0; i < n; ++i)
    in *= a[i];
```

对于你使用的翻译器，在变量 in 的计算结果溢出之前，N 的最大值可以是多少？在一个任意的 C 翻译器下呢？

5.4 阅读下面的代码：

```
long lo = 1.0;
int a[N];

for (i = 0; i < n; ++i)
    lo *= a[i];
```

对于你使用的翻译器，在变量 lo 的计算结果溢出之前，N 的最大值可以是多少？在一个任意的 C 翻译器下呢？

5.5 一个标准 C 的实现可以使 sizeof(long) 等于一个字节吗？这样的实现有什么特殊的属性？

5.6 [**难**] 写一个程序，该程序仅通过算术运算就可以确定 <limits.h> 中定义的宏的值。假设你不知道整数表示的底层实现。

5.7 [**很难**] 修改上题中的程序，使它能在某个实现下安全地工作，该实现遇到整数溢出时会终止程序的执行。假设一旦发生溢出，程序不能重新收回控制权。

第 6 章

<locale.h>

6.1 背景知识

头文件 <locale.h> 是 X3J11 C 标准委员会的一项发明。在 C 的早期实现中几乎没有和区域设置相似的东西，这就和该委员会已宣布的目标“将已经有的实践制定为标准”有些不一致了。然而，当时我们这些 X3J11 委员会成员都认为这种做法的动机是非常好的——这当然是自我辩护了。

历史 这个特殊的头文件是在 C 标准制定工作开始 5 年之后突然出现的。当时，很多人都认为 C 标准的实质性工作已经完成了，我们只是简单地对该产品上做一些收尾的工作。因为这个产品花费了我们 5 年的时间，所以任何性质的更改都会遭到抵抗。

大约就是那个时候，我们发现很多欧洲人对 X3J11 开发的 C 标准中的某些部分不是太满意，它在几个关键的地方太美国化了。他们认为美国人不了解世界市场的需求。因此，他们更愿意等待，希望在一个更合适的场合斗争。因此，欧洲人理所当然地认为 C 的 ISO 标准应该和 ANSI C 标准不同。

我们很多人都不同意那种立场。我们认为不论 ANSI 开发的是什么标准，都要被国际社会认可。以前也有计算机语言在世界各地有不同的标准，我们已经看到了它们的下场。我们感觉到，如果 C 的最终版本来自另一个委员会，而且他们只是参考我们的成果，那么我们 5 年的努力就成了泡影。

所以我们让欧洲人列出了他们想要修改的内容，这些条目中的大部分都是关于让 C 程序能适应不同文化的处理方式。对于那些有很多语言和国家共存的大陆（例如欧洲）来说，这个问题很突出。美国人习惯了单一的语言（即使不是官方的，也被广泛使用）和一个相当简单的字母表。

AT&T Bell 实验室竟然专门召开了一个特殊的会议来讨论国际化（这是一个很长的词，现在人们越来越频繁地讨论它。似乎它没有短一点的同义词来代替。非正式的解决方法是引入 *I18N*，18 代表省略的字母的个数）的各种问题。该会议提出了向 C 标准中加入区域设置支持的建议。最后被人们所接受的机制和最原始的提议相当接近。

把区域设置加到C中取得了预期的效果。很多反对把ANSI C标准作为国际标准的声音都不见了。据我估计，解决区域设置问题又花费了X3J11一年的时间。并且我们可能又花费了一年的时间来处理来自国际社会的剩余问题。（WG14——ISO C标准委员会，现在仍然在做着完善现存的C标准的工作）。然而，我们成功地完成了一个C标准，这个标准的ANSI版本和ISO版本目前是相同的。

环境

编写适应性强的代码并不完全是一个新的话题，早期的形式大约15年前就在UNIX操作系统上出现了。人们想出了一种方法，就是为某些产生新进程的系统调用添加环境变量。（这种服务在UNIX领域被称为exec，或者它的变化形式。）环境变量是一个无限的名字集合，每一个名字都确定一个以空字符结尾的字符串值。我们可以在一个进程中添加、修改或者删除环境变量。如果一个进程引发了另一个新进程，环境变量就会被自动复制到新进程的映象中。

新进程可以忽略环境变量，但这样它会丢失几十或者几百个字节的存储空间。或者它也可以找到某些环境变量并访问它们的当前值。一个常见的变量是"TZ"，它可以为库的日期函数提供关于当前时区的信息。如果"TZ"的值是EST05EDT，时间函数就会把当地的标准时间标为EST，把当地的夏令时标为EDT。当地标准时区比UTC——也就是过去所说的格林尼治平时——早5个小时。

环境变量有很多种用途。它们是将文件名隐式地添加到应用程序中的一种好方法，因为把文件名和程序直接捆绑在一起有很多弊端。如果不考虑那些用户不需要知道的隐藏的文件名，提示用户输入文件名也可以说是一种很好的方法。在命令行中提示输入文件名来启动一个程序也很实用，不过它可能过于繁琐。如果几个程序需要访问一个相同的文件名，就更加复杂了。相比之下，下面介绍的方法就更加简便。为该文件名设置一个环境变量，并将其放在一个启动会话的脚本中。虽然文件名被存放在了一个地方，但是它在整套程序中都可以被使用。

微软的MS-DOS也支持环境变量——这是借鉴UNIX的方法之一。有些商业软件包也把使用环境变量作为一种优势。环境变量最常见的一个作用是查找目录，这里的目录是指包含支持文件的特殊路径或者适合存储临时文件的路径。当然它们也有很多其他的用途。

函数 getenv

C标准库包含了函数getenv，该函数在<stdlib.h>中声明。以一个环境变量为参数调用getenv，假如环境变量的值存在，它就会返回一个指向这个环境变量的值串的指针。引用一个没有定义的环境变量也不会引起错误。

函数 putenv 然而，C 标准库并没有包含和 getenv 相对应的函数 putenv。putenv 可以修改和环境变量相关联的值。如果只是简单的输入，X3J11 不知道怎样描述 putenv 的含义。各种单用户多进程系统中的 putenv 各不相同。所以可以为读取环境变量写出可移植的代码，但不能定义一种标准的方式来修改它们。

为什么引入区域设置 区域设置可以提供哪些环境变量不能提供的东西？一个词：结构。这是一个面向对象的时代，所以，可以把区域设置看成面向对象的环境变量。一个区域设置可以提供很多相联系的参数的信息。这些值对一个具体的文化来说是一致的。要传递同样数量的信息，必须为环境变量在命名空间中保留很多的名字，而且还要为信息子集修改的不一致性承担风险。

顺便说一下，当我提及一种文化时，我并不仅仅是说使用同一种语言的团体。美国人书写日期为 7/4/1776（独立日），同样的日期在英国被写为 4/7/1776（感恩节）。甚至在美国国内，人们的习惯也不同。例如一般人可能把一个借款记为 $-123.45，而会计可能更喜欢（$123.45）。

6

类别 由于这种原因和其他一些原因，区域设置就有了子结构。可以设置整个区域，也可以修改一个或者多个*类别*。头文件 <locale.h> 定义了一些宏，这些宏具有像 LC_COLLATE 和 LC_TIME 这样的名字。每一个宏都可以扩展为一个整数值，你可以把这个整数值作为修改区域设置的函数 setlocale 的类别参数。存在以下几个独立的类别：

- 控制整理顺序（LC_COLLATE）；
- 字符分类（LC_CTYPE）；
- 货币格式（LC_MONETARY）；
- 其他的数字格式（LC_NUMERIC）；
- 时间（LC_TIME）。

一个实现也可以选择提供附加的类别。当然，使用了附加类别的程序，其可移植性就会有所下降。

隐藏在类别背后的思想是应用程序可能希望对它的区域设置做某些小的变动。它可能想用当地的语言打印日期，同时使用这种语言规定的格式。它也可能选择使用点作为十进制的小数点，即使当地的习惯做法是使用逗号。或者应用程序可能会先完全适应一个具体的区域设置，然后改变 LC_MONETARY 类别，以适应全球性的企业财务信息处理标准。

6.2 C 标准的内容

<locale.h>

7.4 区域设置 <locale.h>

头文件 <locale.h> 声明了两个函数和一种类型，并定义了一些宏。

声明的类型是：

struct lconv

```
struct lconv
```

它包含的成员和数字值的格式有关。这个结构至少应包含以下的成员，不分先后顺序。这些成员的语义和它们的取值范围在 7.4.2.1 中有说明。在 "C" 区域设置中，这些成员的值在下面的注释部分有指定。

```
char *decimal_point;        /* "." */
char *thousands_sep;        /* "" */
char *grouping;             /* "" */
char *int_curr_symbol;      /* "" */
char *currency_symbol;      /* "" */
char *mon_decimal_point;    /* "" */
char *mon_thousands_sep;    /* "" */
char *mon_grouping;         /* "" */
char *positive_sign;        /* "" */
char *negative_sign;        /* "" */
char int_frac_digits;       /* CHAR_MAX */
char frac_digits;           /* CHAR_MAX */
char p_cs_precedes;         /* CHAR_MAX */
char p_sep_by_space;        /* CHAR_MAX */
char n_cs_precedes;         /* CHAR_MAX */
char n_sep_by_space;        /* CHAR_MAX */
char p_sign_posn;           /* CHAR_MAX */
char n_sign_posn;           /* CHAR_MAX */
```

NULL

定义的宏有 NULL（7.1.6 中有说明）和下面的宏：

LC_ALL
LC_COLLATE
LC_CTYPE
LC_MONETARY
LC_NUMERIC

```
LC_ALL
LC_COLLATE
LC_CTYPE
LC_MONETARY
LC_NUMERIC
LC_TIME
```

这些宏展开为整数常量表达式，它们的值各不相同，可以作为 setlocale 函数的第一个参数使用。以 LC_ 和一个大写字母开头的其他宏也可以由实现指定[100]。

7.4.1 区域设置控制

setlocale

7.4.1.1 函数 setlocale

概述

```
#include <locale.h>
char *setlocale(int category, const char *locale);
```

说明

函数 setlocale 按照参数 category 和 locale 选择适合程序的区域设置的各个类别。函数 setlocale 可能被用来修改和查询程序当前的整个区域设置或者其中的一部分。参数 category 的值为 LC_ALL，对程序的整个区域设置进行命名，而其他的值只对程序区域设置的一部分进行命名。类别 LC_COLLATE 影响函数 strcoll 和 strxfrm 的行为；类别 LC_CTYPE 影响字符处理函数[101]和多字节函数的行为；类别 LC_MONETARY 影响函数 localeconv 返回的货币格式信息；类别 LC_NUMERIC 影响格式化输入输出函数和字符转换函数的小数点字符，以及函数 localeconv 返回的货币格式信息；类别 LC_TIME 影响函数 strftime 的行为。

参数 locale 的值 "C" 为 C 翻译器指定了最小的环境值 "" 指定了实现定义的本地环境。其他实现定义的串可能作为 setlocale 的第二个参数传递。

在程序启动时，执行下面的代码或者和它等价的代码：

```
setlocale(LC_ALL, "C");
```

实现的行为和没有库函数调用函数 setlocale 的情况相同。

返回值

如果参数 locale 是一个指向串的指针且选择有效，则函数 setlocale 返回一个和新的区域设置指定的 category 相联系的串的指针。如果选择因故无法实现，则函数 setlocale 返回一个空指针且程序的区域设置不改变。

如果参数 locale 是一个空指针，则函数 setlocale 返回一个和程序当前区域设置的 category 相关的串的指针，程序的区域设置不改变[102]。

函数 setlocale 返回的指向串的指针要满足对那个串值和它的相关的类别的后继调用会恢复程序区域设置的相应的部分。指向的串不能被程序修改，但可能会被函数 setlocale 的下一次调用覆盖。

参见：格式化输入输出函数（7.9.6）、多字节字符函数（7.10.7）、多字节字符串函数（7.10.8）、字符串转换函数（7.10.1）、strcoll 函数（7.11.4.3）、strftime 函数（7.12.3.5）、strxfrm 函数（7.11.4.5）。

7.4.2 数字格式习惯查询

localeconv

7.4.2.1 函数 localeconv

概述

```
#include <locale.h>
struct lconv *localeconv(void);
```

说明

函数 localeconv 根据当前区域设置的规则，对一个 struct lconv 类型的对象的成员进行赋值，使其值适合格式化数字量（货币等）。

结构中 char* 类型的成员是指向字符串的指针，它们中的任何一个都（除了 decimal_point）可以指向 ""，以说明该值在当前区域设置下不可用或者长度为零。char 类型的成员是非负数字，任一成员都可以是 CHAR_MAX，以说明该值在当前区域设置下不可用。这些成员有：

char *decimal_point

用来格式化非货币量的小数点字符。

char *thousands_sep

用来对格式化的非货币量中小数点前面的数字进行分组的字符。

char *grouping

一个字符串，它的元素说明了格式化的非货币量中每一组数字的数目。

char *int_curr_symbol

适合当前区域设置的国际货币符号。前 3 个字符包含了符合 ISO 4217:1987 标准的字母国际货币符号。第四个字符（后面紧跟空字符）是国际货币符号与货币量之间的分隔符。

char *currency_symbol

适用于当前区域设置的当地流通符号。

char *mon_decimal_point

用来格式化货币量的小数点。

char *mon_thousands_sep

对格式化的货币量中小数点前面的数字进行分组的分隔符。

char *mon_grouping

一个字符串，它的元素说明了格式化的货币量分组中每组数字的数目。

char *positive_sign

用来说明一个非负的格式化货币量的字符串。

6

char *negative_sign

用来说明一个负的格式化货币量的字符串。

char int_frac_digits

在一个国际标准格式化货币量中显示的小数数字（小数点后面的数字）的数目。

char frac_digits

在一个格式化货币量中显示的小数数字（小数点后面的数字）的数目。

char p_cs_precedes

当 currency_symbol 放在一个非负的格式化货币量的值的前面或者后面时，分别设为 1 或者 0。

char p_sep_by_space

currency_symbol 与非负的格式化货币量的值用空格分开时为 1，否则为 0。

char n_cs_precedes

currency_symbol 放在一个负的格式化货币量的前面或者后面时，分别设为 1 或者 0。

char n_sep_by_space

currency_symbol 与负的格式化货币量用空格分开时为 1，否则为 0。

char p_sign_posn

它的值用来说明一个非负的格式化货币量中 positive_sign 的位置。

char n_sign_posn

它的值用来说明一个负的格式化货币量中 negative_sign 的位置。

grouping 和 mon_grouping 的元素的解释如下：

CHAR_MAX 不能进一步分组；

0　　剩余的数字重复使用前一个元素；

其他值　　整数值是组成当前组的数字的数目。检查下一个元素来确定当前分组前面的下一组数字的数目。

p_sign_posn 和 n_sign_posn 的值解释如下：

0　　数字与 currency_symbol 放在括号中；

1　　符号字符串放在数字与 currency_symbol 前面；

2　　符号字符串放在数字与 currency_symbol 后面；

3　　符号字符串紧挨着放在 currency_symbol 前面；

4　　符号字符串紧挨着放在 currency_symbol 后面。

实现的行为应假设没有库函数调用 localeconv。

返回值

函数 localeconv 返回一个指向已填充对象的指针。返回值指向的结构不能被程序修改，但是可能会被 localeconv 的下一次调用覆盖。另外，用类别 LC_ALL、LC_MONETARY 或者 LC_NUMERIC 调用函数 setlocale 可能会覆盖结构的内容。

例子

下面的表举例说明了可能会被四个国家用来格式化货币量的规则：

国家	正格式	负格式	国际格式
意大利	L.1.234	-L.1.234	ITL.1.234
荷兰	F 1.234,56	F -1.234,56	NLG 1.234,56
挪威	kr1.234,56	kr1.234,56-	NOK 1.234,56
瑞士	SFrs.1,234.56	SFrs.1,234.56C	CHF 1,234.56

对这 4 个国家，localeconv 返回的结构的货币成员的值分别是：

	意大利	荷兰	挪威	瑞士
int_curr_symbol	"ITL."	"NLG"	"NOK"	"CHF"
currency_symbol	"L."	"F"	"kr"	"SFrs."
mon_decimal_point	""	","	","	"."
mon_thousands_sep	"."	"."	"."	","
mon_grouping	"\3"	"\3"	"\3"	"\3"
positive_sign	""	""	""	""
negative_sign	"-"	"-"	"-"	"C"
int_frac_digits	0	2	2	2
frac_digits	0	2	2	2
p_cs_precedes	1	1	1	1
p_sep_by_space	0	1	0	0
n_cs_precedes	1	1	1	1
n_sep_by_space	0	1	0	0
p_sign_posn	1	1	1	1
n_sign_posn	1	4	2	2

脚注

100. 参考“库的展望”（7.13.3）。

101. 7.3 中行为不受当前区域设置影响的仅有的两个函数是 isdigit 和 isxdigit。

102. 当 category 的值为 LC_ALL 时，对一个异构的区域设置，实现必须安排在一个串中对各个类别编码。

6.3 <locale.h> 的使用

区域设置中提供的很多信息都纯粹是信息性质的。例如 C 从来都不会把货币值当作一个特殊的日期类型来处理，所以当类别 LC_MONETARY 改变时，其他的 C 标准库函数都不会受到影响。但另一方面，区域设置的某些变化肯定会影响到某些库函数的行为。如果一种文化使用逗号作为一个十进制数小数点，那么扫描函数就应该能够接受逗号，而打印函数应该把逗号放在适当的位置，这就是实际情况。下面是所有的随着区域设置的改变而变化的库行为。

库的变化

- 当类别 LC_COLLATE 改变时，<string.h> 中声明的函数 strcoll 和 strxfrm 会改变它们比较的方式。
- 当类别 LC_CTYPE 改变时，<ctype.h> 中声明的函数、<stdio.h> 中声明的打印和扫描函数和 <stdlib.h> 中声明的数字转换函数会改变它们测试和修改某些字符的方式。
- 当类别 LC_CTYPE 改变时，<stdlib.h> 中声明的多字节函数和 <stdio.h> 中声明的打印和扫描函数会改变它们解析和转换多字节串的方式。
- 当类别 LC_NUMERIC 改变时，<stdio.h> 中声明的打印和扫描函数与 <stdlib.h> 中声明的 stof 和 strtod 会改变它们对十进制小数点字符的使用方式。
- 当类别 LC_TIME 改变时，<time.h> 中声明的函数 strftime 会改变它把时间转换为字符串的方式。
- 当类别 LC_MONETARY 或者 LC_NUMERIC 改变时，<locale.h> 中声明的函数 localeconv 会改变它的返回值。

这么多的变化还是很可怕的。如果很多C标准库都会改变它的行为，那么怎么能写出可移植的代码呢？能把你的代码放到德国使用吗？当它在那里运行的时候 isalpha 将会做些什么呢？如果把你的代码和另一个源程序的函数混合起来，它们能制造多少麻烦？函数每一次得到控制权的时候，都有可能在不同的区域设置下运行。代码要怎样适应这些情况？

当我们详细地描述了区域设置之后，X3J11 对这些问题很苦恼。我们认识到很多人根本就不关心这种机制，因为添加区域设置可能对那些人影响不太大。还有一些人想完成某些适度的目标，他们想让和美国文化联系在一起的旧的C习惯跟他们的文化一致。还有一些人的目标很高，他们希望写出的程序不经过修改，就可以以目标模块或者可执行程序的形式在众多市场上出售。这些代码中关于改变区域设置的部分一定会很复杂。

"C" 区域设置

使用区域设置最简单的方法就是不去管它们。每一个标准C程序都是在 "C" 区域设置下启动。在这个区域设置中，传统的库函数行为和它们一般的行为相同。例如，islower 只对英语字母表中的 26 个小写字母返回非零值，小数点是一个点。如果你的程序从没有调用过 setlocale，这种行为就不会改变。

本地区域设置

另一种使用区域设置最简单的方法是只在程序启动之后改变一次区域设置，然后保持不变。C标准对 "C" 之外的区域设置的名字没有任何要求。但是它确实定义了由空字符串 "" 指定的*本地区域设置*。如果你的程序执行：

```
setlocale (LC_ALL,"")
```

它就转移到本地区域设置。大概每一个实现都会设计一种方法来确定一个能使当地用户满意的本地区域设置。（当然，一个不在乎区域设置的实现会让本地区域设置和 "C" 区域设置相同。）

区域设置的恢复

一旦区域设置可以改变，在使用库函数的时候就一定要更加小心。某些东西会变得简单，例如显示格式正确的日期或者选择正确的空白字符，但其他的就不那么确定了，例如使用 <ctype.h> 中声明的函数分析字符串。必要时，可以把部分或者全部区域设置恢复为 "C" 区域设置。开始可以写：

```
#include <locale.h>
#include <stdlib.h>
#include <string.h>
    ...
    char *ls = setlocale(LC_CTYPE, "C");
    char *ss = ls ? malloc(strlen(ls) + 1) : NULL;

    if (ss)
        strcpy (ss, ls) ;
```

现在就可以放心地在 "C" 区域设置下使用 <ctype.h> 中声明的函数了。完成的时候，可以通过下面的代码恢复原来的区域设置：

```
setlocale(LC_CTYPE, ss);
free(ss);
```

注意，这些代码在堆用完且 malloc 分配空间失败的时候不能起作用，它只是简单地避免不明智地使用任何空指针。如果能保证对 setlocale 的其他调用不会在上面的两段代码之间起作用，就可以不用对分配空间和复制 setlocale 的返回的区域设置串进行处理。

格式化值

有两个区域设置类别说明了如何对值进行格式化以和当地的习惯相匹配。

- LC_MONETARY 类别说明怎样格式化货币量，既能符合当地习惯，又和国际标准（ISO 4217）一致。
- LC_NUMERIC 规定了 C 标准库使用的小数点符号，也说明了怎样格式化非货币量。

例如，这里给出了依照当地习惯对货币量 $-3.00 格式化的各种方式，它们由存储在 struct lconv 中的 3 个成员的值决定：

n_sep_by_space: 0

n_sign_posn:	0	1	2	3	4
n_cs_precedes:0	(3.00$)	-3.00$	3.00$-	3.00-$	3.00$-
1	($3.00)	-$3.00	$3.00-	-$3.00	$-3.00

n_sep_by_space: 1

n_sign_posn:	0	1	2	3	4
n_cs_precedes:0	(3.00$)	-3.00$	3.00$-	3.00-$	3.00$-
1	($3.00)	-$3.00	$3.00-	-$3.00	$-3.00

这个例子假设结构成员 currency_symbol 指向 "$"，mon_decimal_point 指向 "."，negative_sign 指向 "-"，并且 frac_digits 的值为 2。这个例子不能说明成员 mon_grouping 和 mon_thousands_sep 的影响，这两个成员描述了对小数点左边的数进行分组和分离的方法。

有 3 个附加的成员来描述怎样格式化正的货币量。它们是 p_sep_by_space、p_sign_posn 和 p_cs_precedes。对于国际货币量，成员 int_curr_symbol 决定了货币符号（而不是 currency_symbol），int_frat_digits 决定了显示多少位（而不是 frac_digits）。如果要格式化非货币量，就应该使用成员 decimal_point、grouping 和 thousands_sep。

要了解的东西实在是太多了。也许你可以在整个应用程序的始终都使用这个信息，但也可能不行。各部分在细节方面不是很清楚，我们真正需要的

是一种格式化数值数据的方式，利用这种方式可以在一个地方应用所有相关信息。不幸的是，C 标准并没有定义这样的函数。

函数 _Fmtval

我决定自己定义这个缺失的函数。在经历了几次失败之后，最后我终于可以作以下声明了：

```
char  *_Fmtval(char *buf,double val, int frac_digs):
```

调用者提供字符缓冲区 buf 来存放格式化信息的值。（现代的发展趋势是规定缓冲区的最大长度。如果没有这样的限制检查的话，这个函数会变得相当复杂。）为了简便一点，这个函数返回 buf 的值，然后它把格式化的值作为一个以空字符结束的串存储起来。

还要把待格式化的值 val 指定为 *double* 类型，这是为小数部分服务的，使它至少具有 16 位的精度。对于一个非货币的值，frac_digits 指定了包含在格式化值中的小数部分的位数，struct lconv 的成员对这个参数没有指导意义。

这就是这个设计的精巧之处（也许太精巧了）。区域设置信息表明了值的 4 种不同格式：

- 国际货币量；
- 本地货币量；
- 没有小数点或者小数部分的货币量；
- 有小数点和小数部分的货币量。

只有在第四种情况下才需要提供一个代表小数部分的位数的（非负）值。这就意味着在其他的情况下可以不去理会参数 frac_digits 的各种不同的负值。

图 6-1 显示了文件 xfmtval.c，这个文件定义了函数 _Fmtval。它通过检查 frac_digits 的值来区分这四种不同的格式：

- 值 -2（宏 FN_INT_CUR）告诉函数格式化一个国际货币量。
- 值 -1（宏 FN_LCL_CUR）告诉函数格式化一个区域货币量。
- 其他所有的值都是告诉函数格式化一个非货币量。然而，对一个包含小数点和小数部分的函数来说，小数部分的位数，无论多么确定必须是一个不等于 CHAR_MAX（<limits.h> 中定义）的非负值。所以，如果使用 CHAR_MAX 的值或者其他不等于 -1 或者 -2 的负值调用函数 _Fmtval，就相当于告诉函数格式化一个不含小数点或者小数部分的非货币量。
- 其余的任何一个不等于 CHAR_MAX 的非负值告诉函数格式化一个含小数点和小数部分的非货币量，且这个值指定了小数部分数字的数目。

图 6-1
xfmtval.c

```
/* _Fmtval function */
#include <limits.h>
#include <locale.h>
#include <stdio.h>
#include <string.h>

        /* macros */
#define FN_INT_CUR -2
#define FN_LCL_CUR -1

char *_Fmtval(char *buf, double d, int fdarg)
    {                   /* format number by locale-specific rules */
    char *cur_sym, dec_pt, *grps, grp_sep, *sign;
    const char *fmt;
    int fd, neg;
    struct lconv *p = localeconv();

    if (0 <= d)
        neg = 0;
    else
        d = -d, neg = 1;
    if(fdarg == FN_INT_CUR)
       {                  /* get international currency parameters */
       cur_sym = p->int_curr_symbol;
       dec_pt  = p->mon_decimal_point[0];
       fmt = "$-V";
       fd = p->int_frac_digits;
       grps = p->mon_grouping;
       grp_sep = p->mon_thousands_sep[0];
       sign = neg ? p->negative_sign : p->positive_sign;
       }

else if (fdarg == FN_LCL_CUR)
    {                             /* get local currency parameters */
    static const char *ftab[2][2][5] = {
        {{"(V$)", "-V$", "V$-", "V-$", "V$-"},
         {"($V)", "-$V", "$V-", "-$V", "$-V"}},
        {{"(V $)", "-V $", "V $-", "V- $", "V$-"},
         {"($ V)", "-$ v", "$ v-", "-$ V", "$ -V"}}};

    cur_sym = p->currency_symbol;
    dec_pt = p->mon_decimal_point[0];
    if (neg)
        fmt = ftab[p->n_sep_by_space == 1]
            [p->n_cs_precedes == 1][p->n_sign_posn < 0
            || 4 < p->n_sign_posn ? 0 : p->n_sign_posn];
    else
        fmt = ftab[p->p_sep_by_space == 1]
            [p->p_cs_precedes == 1][p->p_sign_posn <0
            || 4 < p->p_sign_posn ? 0 : p->p_sign_posn];
    fd = p->frac_digits;
    grps = p->mon_grouping;
    grp_sep = p->mon_thousands_sep[0];
    sign = neg ? p->negative_sign : p->positive_sign;
    }
```

图 6-1
(续)

```
    else
        {               /* get numeric parameters(cur_sym not used) */
        dec_pt = p->decimal_point[0];
        fmt = "-V";
        fd = fdarg;
        grps = p->grouping;
        grp_sep = p->thousands_sep[0];
        sign= neg ? "-":"";
        }
     {                  /* build string in buf under control of fmt */
    char *end, *s;
    const char *g;
    size_t i, ns;

    for (s = buf; *fmt; ++fmt, s += strlen(s))
        switch (*fmt)
            {                              /* process a format char */
        case '$':                 /* insert currency symbol string */
            strcpy(s, cur_sym);
            break;
        case '-':                              /* insert sign string */
            strcpy(s, sign);
            break;
        default:                     /* insert literal format char */
        *s++ = *fmt, *s = '\0';
        break;
    case'V':                              /* insert formatted value */
        sprintf(s, "%#.*f",
            0< fd && fd != CHAR_MAX ? fd : 0, d);
        end = strchr(s, p->decimal_point[0]);
        for (ns = 0, i = end - s, g = grps; 0 < i; ++ns)
            {                             /* count separators to add */
            if (g[0] <= 0 || i <= g[0] || g[0] == CHAR_MAX)
                break;
            i -= g[0];
            if (g[1] != 0)
                ++g;
            }
        memmove(end + ns, end, strlen (end) + 1);
        i = end - s, end += ns;
        *end = 0 <= fd && fd != CHAR_MAX ? dec_pt : '\0';
        for (g = grps; 0 < i; --ns)
            {                     /* copy up and insert separators */
            if (g[0] <= 0 || i <= g[0] || g[0] == CHAR_MAX)
                break;
            i -= g[0], end -= g[0];
            memmove(end, end - ns, g[0]);
            *--end = grp_sep;
            if(g[1] != 0)
                ++g;
            }
        }
    }
    return (buf);
    }
```
□

这个函数简单易懂，但它包含了很多枯燥的内容。前半部分简单地整理了符合格式化情况要求的参数集合。它选择一个格式化的串 fmt 把产生的字符送到缓冲区 buf 中。注意，因为区域设置可以改变，所以这些代码认为 struct lconv 的成员的值没有意义。这里使用 <stdio.h> 中的函数 sprintf 把 *double* 类型的值 d 转换到缓冲区中。（这个函数还有很多其他功能）sprintf 中的这个格式串保证了缓冲区中有小数点，如果有数字的话，后面紧接着就是小数的位数。

然后，剩下的工作就是确定要在小数点左边的字符之间插入多少分隔符；然后把这些分隔符插入到这些字符中；然后使用 <string.h> 中声明的函数 memmove 把这些字符转移到缓冲区中。即使源区域和目标区域有重叠，那个函数也会保证复制的正确性。注意，函数把 sprintf（它本身也会随区域设置而变化）产生的小数点用选定格式中的小数点代替。

使用 _Fmtval 要使用函数 _Fmtval，首先必须对它进行声明并且定义和它相关的宏。虽然可以设计一种方法把这些信息包含在头文件中，但我并没有这样做。（参考 6.4 节的讨论。）所以必须加上下面的内容：

```
#define FV_INTEGER -3
#define FV_INT_CUR -2
#define FV_LCL_CUR -1
char *_Fmtval (char *, double, int);
```

把这几行代码放在程序的最上面；或者把它们放在一个单独的头文件中，然后包含这个头文件。这样就可以通过各种不同的方式调用这个函数了。例如下面的代码：

```
#include <stdio.h>
...
    char buf[100];

    printf("You ordered %s sheets,",
        _Fmtval(buf, (double)nitems, FV_INTEGER);
    printf(" each %s square cm.\n",
        _Fmtval(buf, size, 3);
    printf("Please remit %s to our New York office,\n",
        _Fmtval(buf, cost, FV_INT_CUR));
    printf("(that's %s).\n",
        _Fmtval(buf, cost, FV_LCL_CUR));
```

可能会输出以下内容：

```
You ordered 1,340,000 sheets, each 1,204.787 square cm.
Please remit USD 18,279 to our New York office,
(that's $18,278.85).
```

想象一下这种方法：通过直接查看 struct lconv 的内容，来设法产生这样的结果。很明显，函数 _Fmtval 的作用很关键。

宏 NULL 头文件 <locale.h> 也定义了空指针宏 NULL。第 11 章详细地讨论了这个宏。

6.4 <locale.h> 的实现

这一章的代码很多。和前面的章节不同的是，这些代码涉及了 C 标准库的所有部分。在前面的部分，我们已经体会到了函数 _Fmtvl 的复杂。它使用了 <sring.h> 中的字符串操作函数和 <stdio.h> 中的格式输出函数。你会在这些头文件和后面其他的一些头文件中看到这些代码，我们不能对每一个新函数都作很详细的说明，只描述它们的某些特殊用法（例如 sprintf 格式 "%#.*f"）。如果对某个函数不熟悉，可以到后面的某个章节查找相关的信息。

这里有一些指导性的说明。图 6-2 是本章中定义的具有外部连接的函数和数据对象的调用树。我用括号把数据对象的实体标识出来了。每一个外部名字后面是定义它的 C 源文件的名字，以及该源文件所在的页码。每一个函数名下面且向右缩进一个制表符处是该函数用到的所有名字。（后面涉及相同的函数名的子树被省略了。）

例如，函数 setlocale 由 C 源文件 setlocal.c 定义。该函数调用了它自己，也用到了同一个源文件中定义的数据对象 _Clocale，同时它也调用了函数 _Defloc、_Getloc 和 _Setloc。

如果不能理解后面的说明，可以时常回来看看这棵调用树。这对掌握 <locale.h> 中函数的整体结构有很大的帮助。

图 6-2 <locale.h> 的调用树

```
localeconv                              localeco.c,p.  97
setlocale                               setlocal.c,p. 102
    setlocale                           setlocal.c,p. 102
    [_Clocale]                          setlocal.c,p. 102
    _Defloc                             xdefloc.c,p. 105
    _Getloc                             xgetloc.c,p. 104
        _Freeloc                        xfreeloc.c,p. 118
                [_Loctab]               xloctab.c,p. 117
        _Makeloc                        xmakeloc.c,p. 120
            _Locvar                     xlocterm.c,p. 122
            _Locterm                    xlocterm.c,p. 122
                _Skip                   xgetloc.c,p. 104
            _Readloc                    xreadloc.c,p. 115
                [_Loctab]               xloctab.c,p. 117
                _Skip                   xgetloc.c,p. 104
        _Readloc                        xreadloc.c,p. 115
    _Setloc                             xsetloc.c,p. 106
        [_Costate]                      xstate.c,p. 107
        [_Mbcurmax]                     xstate.c,p. 107
        [_Mbstate]                      xstate.c,p. 107
        [_Wcstate]                      xstate.c,p. 107
```

注意这个调用树中没有函数 _Fmtval，因为 C 标准没有这样要求。顺便说一下，C 标准允许添加函数，所以它们当然可以使用像 _Fmtval 这样奇怪的名字，它们甚至可以用像 fmtval 这样的名字来命名。为了整体风格的一致，我选择了一个留给实现人员使用的名字。

C 标准不允许实现对该函数做的事情有：

- 在一个像 <locale.h> 的标准头文件中包含对 fmtval 的声明；
- 在一个标准头文件中定义形如 FV_INT_CUR 的宏名；
- 让任意的 C 标准库函数调用 fmtval。

这些做法都扰乱了为用户保留的命名空间。

函数淘汰（knocking out）

考虑一下那些遵守这些限制的附加的库函数会怎么样。若程序中声明或者调用了那个假想的 fmtval 函数，连接器就会在扫描 C 标准库时因为不恰当地引用了外部名字而包括该函数，若一个程序自己定义了 fmtval，连接器就不会在扫描 C 标准库时包含该函数。因为没有其他的函数会因为这个 fmtval 的出现而受到影响，所以也不会造成任何损失。用户提供的函数就会将附加的库函数淘汰，因此所有可以通过这种方式被淘汰的函数都可以安全地添加到 C 标准库中。

头文件 <locale.h>

不管用什么样的名字，_Fmtval 已经足够了。本章的剩余部分将会讨论实现 C 标准要求的头文件 <locale.h> 提供的服务。

函数 localeconv

实现 <locale.h> 的最简单的部分是函数 localeconv。它要做的全部工作就是返回一个指向描述当前区域设置（或者它的一部分）的结构的指针，该结构的类型为 struct lconv，由 <locale.h> 定义。图 6-3 显示了文件 locale.h，图 6-4 显示了文件 localeco.c。（后者的名字被缩减为 8 个字母是为了满足不同系统的命名限制，正如 0.3 节所解释的。）跟 localeconv 放在一起的是 struct lconv 类型的静态数据对象，其地址是由该函数返回的。注意函数 localeconv 包含了 <locale.h> 中定义的隐式宏。

宏 _NULL

它还是通过包含内部头文件 <yvals.h> 来对头文件 <locale.h> 进行参数化。（参考 3.4 节中对这个头文件的讨论。）它允许实现定义宏 _NULL，因此，NULL 可以通过修改来和具体的实现相匹配。（参考第 11 章中对 NULL 的讨论。）在多数情况下，_NULL 的一个合适的定义为：

```
#define _NULL (void *) 0
```

setlocale 的实现

函数 setlocale 要实现很多功能，调用这个函数的时候，该函数必须根据指定的类别和名字决定选择什么区域设置。它还要找到内存中存在的区域设置，或者从一个文件中读取最新指定的区域设置。（当然，这里描述的是一般的情况。一个最简化的实现只可以识别 "C" 和 " " 区域设置，这两个区域设置也可以相同。）并且它必须返回一个名字，以便以后恢复当前的区域设置。

图 6-3 locale.h

```
/* locale.h standard header */
#ifndef _LOCALE
#define _LOCALE
#ifndef _YVALS
#include <yvals.h>
#endif
        /* macros */
#define NULL _NULL
        /* locale codes */
#define LC_ALL      0
#define LC_COLLATE  1
#define LC_CTYPE    2
#define LC_MONETARY 3
#define LC_NUMERIC  4
#define LC_TIME     5
    /* ADD YOURS HERE */
#define _NCAT       6                       /* one more than last */
        /* type definitions */
struct lconv {
        /* controlled by LC_MONETARY */
    char *currency_symbol;
    char *int_curr_symbol;
    char *mon_decimal_point;
    char *mon_grouping;
    char *mon_thousands_sep;
    char *negative_sign;
    char *positive_sign;
    char frac_digits;
    char int_frac_digits;
    char n_cs_precedes;
    char n_sep_by_space;
    char n_sign_posn;
    char p_cs_precedes;
    char p_sep_by_space;
    char p_sign_posn;
        /* controlled by LC_NUMERIC */
    char *decimal_point;
    char *grouping;
    char *thousands_sep;
    };
        /* declarations */
struct lconv *localeconv(void);
char *setlocale(int, const char *);
extern struct lconv _Locale;
        /* macro overrides */
#define localeconv()    (&_Locale)
#endif
```
□

图 6-4 localeco.c

```
/* localeconv function */
#include <limits.h>
#include <locale.h>

        /* static data */
static char null[] = "";
struct lconv _Locale = {
        /* LC_MONETARY */
    null,                                       /* currency_symbol */
    null,                                       /* int_curr_symbol */
    null,                                       /* mon_decimal_point */
    null,                                       /* mon_grouping */
    null,                                       /* mon_thousands_sep */
    null,                                       /* negative_sign */
    null,                                       /* positive_sign */
    CHAR_MAX,                                   /* frac_digits */
    CHAR_MAX,                                   /* int_frac_digits */
    CHAR_MAX,                                   /* n_cs_precedes */
    CHAR_MAX,                                   /* n_sep_by_space */
    CHAR_MAX,                                   /* n_sign_posn */
    CHAR_MAX,                                   /* p_cs_precedes */
    CHAR_MAX,                                   /* p_sep_by_space */
    CHAR_MAX,                                   /* p_sign_posn */
        /* LC_NUMERIC */
    ".",                                        /* decimal_point */
    null,                                       /* grouping */
    null};                                      /* thousands_sep */

struct lconv * (localeconv)(void)
    {                               /* get pointer to current locale */
    return (&_Locale);
    }
```
□

混合的区域设置

最后一个功能是最难的，因为有时候可能会构建一个混合的区域设置，这个区域设置包括各种不同的区域设置类别。例如，可以像这样编写代码：

```
#include <locale.h>
...
    char *s1, s2;

    setlocale(LC_ALL, "");
    s1 = setlocale(LC_CTYPE, "C");
    if ((s2 = malloc(strlen(s1) + 1)))
        strcpy(s2, s1);
```

第一次调用选择要切换到本地区域设置——某一个适合本地操作环境的区域设置。第二次调用恢复为 "C" 区域设置。必须对 s1 指向的串进行备份，因为中间调用 setlocale 可能会修改这个串。如果后面做了以下调用：

```
setlocale(LC_ALL, s2);
```

区域设置就会恢复到以前的混合状态。

区域设置的名字　setlocale 必须设定一个名字，以后它可以使用这个名字来重建各种混合类别的区域设置。C 标准没有说明怎么做，也没有规定这些名字的格式，只说了一个实现必须这样做。

我使用的方案是如果一个区域设置包含了混合类别，就在它的名字中加入限定词。例如，假设基本的区域设置是 "USA"（这个区域设置使用美国的日期等格式）。一个应用程序想让类别 LC_MONETARY 适应 "acct" 区域设置（这个区域设置的记账习惯很特殊），那么这个混合的区域设置名字就是："USA;monetary:acct"。

分号用来把混合区域设置名字的各部分分隔开。在每一部分中，冒号用来把类别名字和它的区域设置名字分隔开。基本区域设置没有类别限定词，当 setlocale 构建一个名字时，它只添加那些和基本区域设置不同的类别。

实现 setlocale 和它的后续函数要做的工作比图 6-2 中的函数的子树要多。它需要各种宏、类型定义以及所有的函数和数据对象的声明。其实这就是头文件的作用，即用来存储所有相互协作的函数之间共享的信息的“仓库”。

然而，那个仓库不能是 <locale.h>，因为我们只需要在某个特定的时期在 <locale.h> 中包含 setlocale 的声明，所有其他的函数声明紧随其后。我的习惯做法是只在标准头文件中包含那些外部可见的名字。头文件 <locale.h> 还声明了数据对象 _Locale，那是因为 localeconv 的屏蔽宏（masking macro）引用了 _Locale。这样的一个头文件已经很完备了。

头文件 "xlocale.h"　我创建了内部头文件 "xlocale.h" 来放置其他的东西。本章其余的 C 源文件包含了这个内部头文件和使用的 C 标准库中的所有其他函数所在的标准头文件。反过来，"xlocale.h" 又包含了 <locale.h>，它也包含了两个其他的内部头文件。"xlocale.h" 中的大部分信息的作用现在都不能体现出来。因此，在本章后面才给出它的全部内容，前面只是出示了 "xlocale.h" 中那些有用的零碎的东西。

类型 _Linfo　第一个要说的是一个数据结构，它包含一个区域设置的全部信息，所以，它一定要包含 struct lconv 的一个实例；它也包含一些指针，这些指针指向 <ctype.h> 中的函数 _Ctype，_Tolower 和 _Toupper 使用的表；它还包含了 C 标准库的其他部分的一些信息。可以说，它就是一个大杂烩。"xlocale.h" 定义了一个叫做 _Linfo 的数据类型，其形式如下：

```
typedef struct _Linfo {
    const char *_Name;   /* must be first */
    struct _Linfo *_Next;
        /* controlled by LC_COLLATE */
    _Statab _Costate;
        /* controlled by LC_CTYPE */
```

```
const short *_Ctype;
const short *_Tolower;
const short *_Toupper;
unsigned char _Mbcurmax;
_Statab '_Mbstate;
_Statab _Wcstate;
    /* controlled by LC_MONETARY and LC_NUMERIC */
struct lconv _Lc;
    /* controlled by LC_TIME */
_Tinfo _Times;
} _Linfo;
```

最初，这个结构中只存在一个实例——setlocal.c 中定义的数据对象 _Clocale。_Clocale 的成员 _Name 有一个非零的初始值，它指向区域设置名字：串 "C"。（这就是该名在结构体中第一次出现的地方。）对 setlocale 的第一次调用所做的是在区域设置改变之前把区域设置指定的信息复制到这个数据对象中。后来的调用就可以简单地通过把相关的信息复制出来而恢复到 "C" 区域设置。

如果函数 _Getloc 想要读入一个新的区域设置（就像本章后面所描述的那样），这个函数就会为一个新的 _Linfo 实例分配空间，并且把 _Clocale 复制进去。然后 _Getloc 读入这个区域设置的所有改变。如果所有的改变都是有效的，函数就把这个新的区域设置添加到一个以 _Clocale._Next 为表头的可变的区域设置链表中。链表中的一个结点的成员 _Next 为一个空指针，这个结点为该链表的表尾。（注意，_Linfo 在这个声明中既作为类型名，又作为结构标签。只有拥有一个标签名的结构才能包含一个指针，这个指针指向这个结构的另一个实例）

类型 _Statab

结构 _Linfo 包含了几个 _Statab 类型的成员。在这个 C 标准库的实现中，某些函数使用*状态表*来定义它们的行为。这就为提高性能提供了最大的灵活性。它也允许使用和 <ctype.h> 中的转换表相似的符号来指定这些函数在一个区域设置中的行为。下面是受到影响的几个函数。

- <string.h> 中声明的 strcoll 和 strxfrm，为定义一个整理顺序把一个字符串转换为另一个字符串。
- <stdlib.h> 中声明的 mbtowc 和 mbstowcs，把一个多字节串转换为一个宽字节字符串。
- <stdlib.h> 中声明的 wctomb 和 wcstombs，把一个宽字节字符串转换为一个多字节串。

头文件 "xstate.h"

我会在后面的章节中描述这些函数的行为。现在，我发现内部头文件 "xstate" 定义了类型 _Statab 和几个有用的宏，并且声明了 _Statab 类型的各种数据对象。内部头文件 "xlocale.h" 包含了头文件 "xstate.h"，以便区域设置改变时获得操作状态表所需的信息。图 6-5 显示了头文件 xstate.h。

6

图 6-5 xstate.h

```
/* xstate.h internal header */
        /* macros for finite state machines */
#define ST_CH       0x00ff
#define ST_STATE    0x0f00
#define ST_STOFF    8
#define ST_FOLD     0x8000
#define ST_INPUT    0x4000
#define ST_OUTPUT   0x2000
#define ST_ROTATE   0x1000
#define _NSTATE     16
        /* type definitions */
typedef struct {
    const unsigned short *_Tab[_NSTATE];
    } _Statab;
        /* declarations */
extern _Statab _Costate, _Mbstate, _Wcstate;
```

类型 _Tinfo

类似地，<time.h> 中声明的函数也有区域设置指定的行为。结构类型 _Tinfo 包含了几个指向以空字符结尾的字符串的成员。这些串用来控制时间函数输出和转换日期和时间的方式。

头文件 "xtinfo.h"

内部头文件 "xtinfo.h" 定义了类型 _Tinfo 并声明了 _Tinfo 类型的数据对象 _Times，这个数据对象保存了关于时间的当前信息。"xlocale.h" 也包含了 "xtinfo.h"，以便区域设置改变时获得操作时间信息所需的信息。图 6-6 显示头文件 xtinfo.h。

现在可以来看一下函数 setlocale 都做了些什么。图 6-7 显示了文件 setlocal.c。它的主要功能是分析一个名字来决定每个类别使用哪种区域设置。另外一个主要功能是构建一个 setlocale 以后可以识别的名字。相比之下，其他的东西就显得不是那么重要了。

图 6-6 xtinfo.h

```
/* xtinfo.h internal header */

        /* type definitions */
typedef struct {
    const char *_Ampm;
    const char *_Days;
    const char *_Formats;
    const char *_Isdst;
    const char *_Months;
    const char *_Tzone;
    } _Tinfo;
        /* declarations */
extern _Tinfo _Times;
```

图 6-7
setlocal.c

```
/* setlocale function */
#include <ctype.h>
#include <string.h>
#include "xlocale.h"

#if _NCAT != 6
#error WRONG NUMBER OF CATEGORIES
#endif
        /* static data */
_Linfo _Clocale = {"C"};
static char *curname = "C";
static char namalloc = 0;                          /* curname allocated */
static const char * const nmcats[_NCAT] = {
    NULL, "collate:", "ctype:", "monetary:",
    "numeric:", "time:"};
static _Linfo *pcats [_NCAT] = {
    &_Clocale, &_Clocale, &_Clocale, &_Clocale,
    &_Clocale, &_Clocale };

char *(setlocale)(int cat, const char *lname)
    {                                                /* set new locale */
    size_t i;

    if (cat < 0 || _NCAT <= cat)
        return (NULL);                               /* bad category */
    if (lname == NULL)
        return (curname);
    if (lname[0] == '\0')
        lname = _Defloc();
    if (_Clocale._Costate._Tab[0] == NULL)
        {                                          /* fill in "C" locale */
        _Clocale._Costate = _Costate;
        _Clocale._Ctype = _Ctype;
        _Clocale._Tolower = _Tolower;
        _Clocale._Toupper = _Toupper;
        _Clocale._Mbcurmax = _Mbcurmax;
        _Clocale._Mbstate = _Mbstate;
        _Clocale._Wcstate = _Wcstate;
        _Clocale._LC = _Locale;
        _Clocale._Times = _Times;
        }                                            /* set categories */
    {
    _Linfo *p;
    int changed = 0;

    if (cat != LC_ALL)
        {                                    /* set a single category */
        if((p = _Getloc(nmcats[cat], lname)) == NULL)
            return (NULL);
        if (p != pcats [cat])
            pcats[cat] = _Setloc(cat, p), changed = 1;
        }
    else
        {                                        /* set all categories */
```

图6-7（续）

```
        for (i = 0; ++i < _NCAT;)
            {                                    /* set a category */
            if ((p = _Getloc(nmcats[i], lname)) == NULL)
                {                       /* revert all on any failure */
                setlocale(LC_ALL, curname);
                return (NULL);
                }
            if (p != pcats[i])
                pcats[i] = _Setloc(i, p), changed = 1;
            }
        if((p = _Getloc("", lname)) != NULL)
            pcats[0] = p;         /* set only if LC_ALL component */
        }
    if (changed)
        {                                     /* rebuild curname */
        char *s;
        size_t n;
        size_t len = strlen(pcats[0]->_Name);

        for (i = 0, n = 0; ++i < _NCAT; )
            if (pcats[i] != pcats[0])
                {                   /* count a changed subcategory */
                len += strlen(nmcats[i])
                    + strlen(pcats[i]->_Name) + 1;
                ++n;
                }
        if (n == 0)
            {                                    /* uniform locale */
            if (namalloc)
                free(curname);
            curname = (char *)pcats[1]->_Name, namalloc = 0;
            }
        else if ((s = (char *)malloc(len + 1)) == NULL)
            {                   /* may be rash to try to roll back */
            setlocale(LC_ALL, curname);
            return (NULL);
            }
        else
            {                                /* build complex name */
            if (namalloc)
                free(curname);
            curname = s, namalloc = 1;
            s += strlen(strcpy(s, pcats[0]->_Name));
            for (i = 0; ++i < _NCAT; )
                if (pcats[i] != pcats[0])
                    {                           /* add a component */
                    *s++ = ';';
                    s += strlen(strcpy(s, nmcats[i]));
                    s += strlen(strcpy(s, pcats[i]->_Name));
                    }
            }
        }
    }
    return (curname);
    }
```

□

setlocale 包含了一些代码，这些代码在第一次尝试改变区域设置的时候把信息复制到 "C" 区域设置中。我采用了这种方法来避免麻烦的雪球效应。把各种区域设置指定的表装入一个结构中很容易。然而，这样做不管用到多少东西都会得到整个雪球。我认为让 setlocale 多做点工作来避免这个问题会更好。你肯定不希望只是在使用函数 isspace 的时候就插入 10KB 的代码。

函数 _Getloc 函数 _Getloc 用来确定内存中是否存在一个和给定类别对应的区域设置。如果不存在，则 _Getloc 通过读取一个区域设置文件来寻找它。下面会详细地讨论这个文件的读取。图 6-8 显示了定义该函数的文件 xgetloc.c。

函数 _Skip C 源文件 xgetloc.c 也定义了函数 _Skip。有几个读取区域设置文件的函数调用 _Skip 来跳过一个字符（而不是空字符）和后面紧跟的空白。这里，空白由空白符和水平制表符组成。使用 _Skip 可以统一区域设置文件中对空白的定义，也可以简化后面的很多代码。

函数 _Defloc 图 6-9 显示了源文件 xdefloc.c。它定义了确定本地区域设置名字的函数 _Defloc。为了确定那个名字，我选择使用环境变量 "LOCALE"。这和确定时区的环境变量 "TZ" 很相似。在程序执行过程中，_Defloc 最多检查环境变量 LOCALE 一次。

函数 _Setloc 图 6-10 显示了文件 xsetloc.c。它定义了函数 _Setloc，这个函数实际上是把新的信息复制到受区域设置改变影响的静态数据的各个位中。（注意它也执行了一点检查更关键的值的功能。）因此，调用 setlocale 就会引入所有这样的信息。我不知道怎样避免这种特殊的雪球。但至少如果把区域设置放到一边不管，就可以避免了。

状态表 为了使文档完整，这里给出了初始的状态表，因为 setlocale 和 _Setloc 都对状态表进行了操作。（时间信息 _Times 存储在文件 asctime.c 中，见 15.4 节。）图 6-11 显示了文件 xstate.c。不要想深入地理解它。现在，我只能说下面给出的这个状态表适用于所有使用状态表的函数。它巧妙的设计能产生对所有这些函数有用的（如果简单的话）结果。它也为在区域设置文件中定义状态表开了个好头。

到目前为止，我已经介绍了支持区域设置的所有基本的机制。这些机制对于在库中构建附加的区域设置已经足够了。只要添加类型 struct lconv 的静态声明并对它们进行合适的初始化就可以了。但是要保证 _Clocale._Next 指向你增加的链表。

区域设置文件 然而，区域设置的真正乐趣是定义一个无限集合的前景。为了实现它，需要在不改变 C 代码的前提下指定一个区域设置。这就要用到前面的图 6-2 中还没有说明过的所有剩余机制。在讲述这种机制之前，必须要先讲一下区域设置文件。

图 6-8
xgetloc.c

```
/* _Getloc and_Skip functions */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "xlocale.h"

const char *_Skip(const char *s)
    {                               /* skip next char plus white-space */
    return (*s == '\0' ? s : s + 1 + strspn(s + 1, "\t"));
    }

_Linfo *_Getloc(const char *nmcat, const char *lname)
    {                   /* get locale pointer, given category and name */
    const char *ns, *s;
    size_t nl;
    _Linfo *p;

      {                             /* find category component of name */
    size_t n;

    for (ns =NULL, s = lname; ; s += n + 1)
        {                              /* look for exact match or LC_ALL */
        if (s[n = strcspn(s, ":;")] == '\0' || s[n] = ';')
        {                                       /* memorize first LC_ALL */
            if (ns == NULL)
                ns = s, nl = n;
            if (s[n] == '\0')
                break;
            }
        else if (memcmp(nmcat, s, ++n) == 0)
            {                               /* found exact category match */
            ns = s + n, nl = strcspn(ns, ";");
            break;
            }
        else if (s[n += strcspn(s + n, ";")] == '\0')
            break;
        }
    if (ns == NULL)
        return (NULL);                                  /* invalid name */
    }
    for (p = &_Clocale; p; p = p->_Next)
        if (memcmp(p->_Name, ns, nl) == 0
            && p->_Name[nl] == '\0')
            return (p);
    {                                       /* look for locale in file */
    char buf [MAXLIN], *s1;
    FILE *lf;
    _Locitem *q;
    static char *locfile;                           /* locale file name */

    if (locfile)
        ;
    else if ((s = getenv("LOCFILE")) == NULL
        || ((locfile = malloc(strlen(s) +1))) == NULL)
        return (NULL);
```

图 6-8
(续)

```
    else
        strcpy(locfile, s);
    if ( (lf = fopen(locfile, "r")) == NULL)
        return (NULL);
    while ((q = _Readloc(lf, buf, &s)) != NULL)
        if (q->_Code == L_NAME
           && memcmp(s, ns, nl) == 0
           && *_Skip(s + nl - 1) == '\0')
           break;
    if (q == NULL)
        p = NULL;
    else if ((p = malloc(sizeof (_Linfo))) == NULL)
        ;
    else if ((s1 = malloc(nl + 1)) == NULL)
         free(p), p = NULL;
    else
        {                                       /* build locale */
        *p = _Clocale;
        p->_Name = memcpy(s1, ns, nl);
        s1[nl] = '\0';
        if (_Makeloc(lf, buf, p))
            p->_Next = _Clocale._Next, _Clocale._Next = p;
        else
            {               /* parsing error reading locale file */
            fputs(buf, stderr);
            fputs("\n-- invalid locale file line\n", stderr);
            _Freeloc(p);
            free(p), p = NULL;
            }
        }
    fclose(lf);
    return(p);
     }
    }
```

图 6-9
xdefloc.c

```
/* _Defloc function */
#include <stdlib.h>
#include <string.h>
#include "xlocale.h"

const char *_Defloc(void)
    {                               /* find name of default locale */
    char *s;
    static char *defname = NULL;

    if (defname)
        ;
    else if ((s = getenv("LOCALE")) != NULL
        && (defname = malloc(strlen(s) + 1)) != NULL)
        strcpy(defname, s);
    else
        defname = "C";
    return (defname) ;
    }
```

图 6-10
xsetloc.c

```
/* _Setloc function */
#include <ctype.h>
#include <limits.h>
#include "xlocale.h"

_Linfo *_Setloc(int cat, _Linfo *p)
    {                                   /* set category for locale */
    switch(cat)
        {                                       /* set a category */
    case LC_COLLATE:
        _Costate = p->_Costate;
        break;
    case LC_CTYPE:
        _Ctype = p->_Ctype;
        _Tolower = p->_Tolower;
        _Toupper = p->_Toupper;
        _Mbcurmax = p->_Mbcurmax <= MB_LEN_MAX
            ? p->_Mbcurmax : MB_LEN_MAX;
        _Mbstate = p->_Mbstate;
        _Wcstate = p->_Wcstate;
        break;
    case LC_MONETARY:
        _Locale.currency_symbol = p->_Lc.currency_symbol;
        _Locale.int_curr_symbol = p->_Lc.int_curr_symbol;
        _Locale.mon_decimal_point = p->_Lc.mon_decimal_point;
        _Locale.mon_grouping = p->_Lc.mon_grouping;
        _Locale.mon_thousands_sep = p->_Lc.mon_thousands_sep;
        _Locale.negative_sign = p->_Lc.negative_sign;
        _Locale.positive_sign = p->_Lc.positive_sign;
        _Locale.frac_digits = p->_Lc.frac_digits;
        _Locale.int_frac_digits = p->_Lc.int_frac_digits;
        _Locale.n_cs_precedes = p->_Lc.n_cs_precedes;
        _Locale.n_sep_by_space = p->_Lc.n_sep_by_space;
        _Locale.n_sign_posn = p->_Lc.n_sign_posn;
        _Locale.p_cs_precedes = p->_Lc.p_cs_precedes;
        _Locale.p_sep_by_space = p->_Lc.p_sep_by_space;
        _Locale.p_sign_posn = p->_Lc.p_sign_posn;
        break;
    case LC_NUMERIC:
        _Locale.decimal_point = p->_Lc.decimal_point[0] != '\0'
            ? p->_Lc.decimal_point : ".";
        _Locale.grouping = p->_Lc.grouping;
        _Locale.thousands_sep = p->_Lc.thousands_sep;
        break;
    case LC_TIME:
        _Times= p->_Times;
        break;
        }
    return(p);
    }
```
□

图 6-11 xstate.c

```
/*_Costate,_Mbstate, and _Wcstate generic tables */
#include <limits.h>
#include "xlocale.h"
#if UCHAR_MAX != 255
#error WRONG STATE TABLE
#endif

        /* macros */
#define X   (ST_FOLD|ST_OUTPUT|ST_INPUT)

        /* static data */
static const unsigned short tab0[257] = {0, /* alloc flag */
X|0X00, X|0x01, X|0x02, X|0x03, X|0x04, X|0x05, X|0x06, X|0x07,
X|0x08, X|0x09, X|0x0a, X|0x0b, X|0x0c, X|0x0d, X|0x0e, X|0x0f,
X|0x10, X|0x11, X|0x12, X|0x13, X|0x14, X|0x15, X|0x16, X|0x17,
X|0x18, X|0x19, X|0x1a, X|0x1b, X|0x1c, X|0x1d, X|0x1e, X|0x1f,
X|0x20, X|0x21, X|0x22, X|0x23, X|0x24, X|0x25, X|0x26, X|0x27,
X|0x28, X|0x29, X|0x2a, X|0x2b, X|0x2c, X|0x2d, X|0x2e, X|0x2f,
X|0x30, X|0x31, X|0x32, X|0x33, X|0x34, X|0x35, X|0x36, X|0x37,
X|0x38, X|0x39, X|0x3a, X|0x3b, X|0x3c, X|0x3d, X|0x3e, X|0x3f,
X|0x40, X|0x41, X|0x42, X|0x43, X|0x44, X|0x45, X|0x46, X|0x47,
X|0x48, X|0x49, X|0x4a, X|0x4b, X|0x4c, X|0x4d, X|0x4e, X|0x4f,
X|0x50, X|0x51, X|0x52, X|0x53, X|0x54, X|0x55, X|0x56, X|0x57,
X|0x58, X|0x59, X|0x5a, X|0x5b. X|0x5c, X|0x5d, X|0x5e, X|0x5f,
X|0x60, X|0x61, X|0x62, X|0x63, X|0x64, X|0x65, X|0x66, X|0x67,
X|0x68, X|0x69, X|0x6a, X|0x6b, X|0x6c, X|0x6d, X|0x6e, X|0x6f,
X|0x70, X|0x71, X|0x72, X|0x73, X|0x74, X|0x75, X|0x76, X|0x77,
X|0x78, X|0x79, X|0x7a, X|0x7b, X|0x7c, X|0x7d, X|0x7e, X|0x7f,

X|0x80, X|0x81, X|0x82, X|0x83, X|0x84, X|0x85, X|0x86, X|0x87,
X|0x88, X|0x89, X|0x8a, X|0x8b, X|0x8c, X|0x8d, X|0x8e, X|0x8f,
X|0x90, X|0x91, X|0x92, X|0x93, X|0x94, X|0x95, X|0x96, X|0x97,
X|0x98, X|0x99, X|0x9a, X|0x9b, X|0x9c, X|0x9d, X|0x9e, X|0x9f,
X|0xa0, X|0xa1, X|0xa2, X|0xa3, X|0xa4, X|0xa5, X|0xa6, X|0xa7,
X|0xa8, X|0xa9, X|0xaa, X|0xab, X|0xac, X|0xad, X|0xae, X|0xaf,
X|0xb0, X|0xb1, X|0xb2, X|0xb3, X|0xb4, X|0xb5, X|0xb6, X|0xb7,
X|0xb8, X|0xb9, X|0xba, X|0xbb, X|0xbc, X|0xbd, X|0xbe, X|0xbf,
X|0xc0, X|0xc1, X|0xc2, X|0xc3, X|0xc4, X|0xc5, X|0xc6, X|0xc7,
X|0xc8, X|0xc9, X|0xca, X|0xcb, X|0xcc, X|0xcd, X|0xce, X|0xcf,
X|0xd0, X|0xd1, X|0xd2, X|0xd3, X|0xd4, X|0xd5, X|0xd6, X|0xd7,
X|0xd8, X|0xd9, X|0xda, X|0xdb, X|0xdc, X|0xdd, X|0xde, X|0xdf,
X|0xe0, X|0xe1, X|0xe2, X|0xe3, X|0xe4, X|0xe5, X|0xe6, X|0xe7,
X|0xe8, X|0xe9, X|0xea, X|0xeb, X|0xec, X|0xed, X|0xee, X|0xef
X|0xf0, X|0xf1, X|0xf2, X|0xf3, X|0xf4, X|0xf5, X|0xf6, X|0xf7,
X|0xf8, X|0xf9, X|0xfa, X|0xfb, X|0xfc, X|0xfd, X|0xfe, X|0xff,
    };

char _Mbcurmax = 1;

_Statab _Costate = {&tab0[1]};
_Statab _Mbstate = {&tab0[1]};
_Statab _Wcstate = {&tab0[1]};
```
□

一个区域设置应该很容易定义。各种各样的人都有机会定义一个区域设置的部分或者全部。不同的人群可能希望：

- 使用当地的习惯和当地的语言打印日期和时间；
- 改变用来读取、转换和输入浮点值的小数点；
- 指定当地货币格式和符号；
- 指定特殊的比较顺序；
- 在 <ctype.h> 中声明的函数定义的字符分类中添加字符、标点符号或者控制符号；
- 改变多字节和宽字节字符的编码方式。

这些改变大致是按照复杂性递增的顺序列出的。几乎每个人都可能想把月份和星期的名字改变为另一种语言。某些人可能会定义一个特殊的整理顺序。而只有最勇敢的人才会考虑修改成一种新的多字节字符编码方式（首先，它可能与翻译器产生的字符串字面量和字符常量不一致）。不过，这些操作中没有一个要求 C 标准库作出改变。

因此，我们的目标是设计一种方法。通过这种方法，普通人也可以定义一个新的区域设置并在程序运行时引入到程序中。当然，这个程序一定会在某种条件下调用 setlocale，并且一定要使用这次调用所修改的信息。在给定了这些明显的先决条件之后，C 标准库应该有助于程序和用户在区域设置规范方面保持一致。

采用的方法是引入两个环境变量和一个文件格式。环境变量如下。

"LOCALE"

- "LOCALE"（本节前面的“函数 _Defloc”部分有讲）：它指定了一个本地区域设置的名字，这个名字是由 setlocale(LC_ALL,"") 这样的调用被选择。

"LOCFILE"

- "LOCFILE"：如果 setlocale 遇到了一个内存中不存在的区域设置名字，它就指定一个可以使用的区域设置文件的名字。

文件格式指定了准备文本文件的方法，这样它就可以定义你想添加的所有附加的区域设置。

例如，一个叫做 xxx 的程序开始先调用上面的 setlocale(LC_ALL,"")。在 MS-DOS 下，可以从一个批处理文件中调用它：

```
set LOCFILE=c:\locales\mylocs.loc
set LOCALE=USA
xxx
```

这会让程序 xxx 读取文件 c:\locales\mylocs.inc 来搜索区域设置名 "USA"。假设程序可以找到该区域设置并且成功地读入，那么程序 xxx 就可以采用适合 "USA" 区域设置的行为来执行。在批处理脚本中把 "USA" 改为 "France"，程序就会在同一个文件中找出一个不同的区域设置。或者可以修改由 "LOCFILE" 指定的文件名，然后总是搜索一般的 "native" 区域设置。这两种方法都是修改本地区域设置的明智的方法。

一个更复杂的程序可能不仅仅使用本地区域设置，它可能会通过各种不同的方式确定类别和区域设置的名字，然后让 setlocale 把它们从区域设置文件中找出来。而且程序在运行的时候，甚至可能重新写入区域设置文件的内容，以迅速建立新的区域设置。如果以上任一种情况发生了，要尽量推迟区域设置和程序绑定的时间。

区域设置文件格式

一个区域设置由各种数据类型组成。一些是数字值，一些是字符串，还有一些是变化格式的表。区域设置中的每一个实体都要有一个不同的名字。当写入区域设置文件来指定希望重新定义的实体时，就会用到这些名字。对于 struct lconv 中的成员，使用成员名作为区域设置文件中的实体名。其他情况下，必须自己为实体命名。

区域设置文件被组织为一个文本行的序列。例如，可以用下面一行开始 "USA" 区域设置的定义：

```
LOCALE USA
```

以后的每一行都以一个预先定义的列表中的关键字开头。使用 NOTE 开始一行注释，使用 SET 来指定一个大写字母的值，就像下面的一样：

```
NOTE   The following sets D(elta) to 'a'-'A'
SET    D   'a' - 'A'
```

然后就可以使用 D 作为表达式中的一个项。

如果关键字是一个实体名，可以在这一行的剩余地方指定它的值，例如：

```
currency_symbol    $
int_curr_symbol    "USD"
frac_digits        2
```

一个串值周围的引号是可选的。只有在需要把空白作为串的一部分的时候才有必要使用引号。可以在任何要求数字值的地方写一个相当复杂的表达式。本节后面会详细地讨论表达式。

每一个新的区域设置的初始值都能和 "C" 区域设置中的相匹配。这样就节省了很多输入的工作。真正需要指定的是 "C" 区域设置中改变的东西。只有当需要一个区域设置的更多完整文档的时候，才可能输入更多的东西。

数字值

需要为 struct lconv 的某些成员指定数字值。这些值包括下面那些 LC_MONETARY 类别的信息：

```
frac_digits
int_frac_digits
n_cs_precedes
n_sep_by_spaces
n_sign_posn
p_cs_precedes
p_sep_by_spaces
p_sign_posn
```

6

这些信息中的每一个都占据一个 *char* 类型成员。<limits.h> 中定义的 CHAR_MAX 的值说明它没有提供任何有意义的值。

<locale.h> 中定义的宏 MB_CUR_MAX 的值，可以随着 LC_CTYPE 类别而改变。我采用实体名

```
mb_cur_max
```

作为保存这个宏值的 *char* 类型的数据对象的名字。

字符串值

需要为 struct lconv 的某些成员指定字符串。这些成员包括 LC_MONETARY 类别的信息

```
currency_symbol
int_curr_symbol
mon_decimal_point
mon_thousands_sep
negative_sign
positive_sign
```

和 LC_NUMERIC 类别的信息

```
decimal_point
thousands_sep
```

顺便说一下需要注意的地方，C 标准认为 mon_decimal_point、mon_thousands_sep、decimal_point 和 thousands_sep 都是长度为 1 的字符串。无论它们是什么，这个实现中的函数都使用每个串的第一个字符。

数字串

需要为 struct lconv 的某些成员指定数字串，它们包括：

```
grouping         (LC_NUMERIC)
mon_grouping     (LC_MONETARY)
```

每一个字符的值指定了十进制小数点左边要分组的字符数。一个零值使串结束并且导致最后一个分组值重复不确定次。一个 CHAR_MAX 值也会终止字符串，且说明分组结束。例如，为了先用 2 然后使用 5 对数字进行分组，就会想到创建数组 {2,5,CHAR_MAX}。然而，在区域设置文件中，可以写：

```
mon_grouping 25^
```

对于数字串，每一个十六进制数字都被它的数字值代替。插入符（^）被 CHAR_MAX 代替。

我引入了少数几个附加的串来指定类别 LC_TIME 的信息。（参考图 6-6 定义的类型 _Tinfo。）其中每个串都被分为几个域。我想不出任何字符可以当作一个通用的域界定符使用，所以我采用通常的做法：串的第一个字符界定第一个域的开始。该字符也界定每一个子域的开始。这样选择的字符不会和域中的任何字符冲突。

作为一个例子，am_pm 实体详细说明了 <time.h> 中声明的函数 strftime 对于 AM/PM 指示符打印的内容。这个串的常用定义是：AM:PM。冒号界定了每一个域的开始。

下面是 LC_TIME 类别的实体名和对英语国家来说某些合理的串值。它们大部分都代表它们自己的含义：

```
    am_pm          :AM:PM
    days           :Sun:Sunday:Mon:Monday:Tue:Tuesday\
Wed:Wednesday:Thu:Thursday:Fri:Friday:Sat:Saturday
    dst_rules      :032402:102702
    time_formats   "|%b %D %H:%M:%S %Y|%b %D %Y|%H:%M:%S"
    months         :Jan:January:Feb:February:Mar:March\
Apr:April:May:May:Jun:June\
Jul:July:Aug:August:Sep:September\
Oct:October:Nov:November:Dec:December
    time_zone      :EST:EDT:+0300
```

注意，可以使用反斜线来使一行延续。包括所有延续的内容，一行最多可以容纳 255 个字符。

字符串 time_formats 指定了 strftime 生成区域设置特有的日期和时间（%c）、日期（%x）以及时间（%c）所使用的格式。我会在第 15 章中进一步讨论这些函数。

"TIMEZONE"
"TZ"

time_zone 的第三个域对 UTC（格林尼治平时）的分钟而不是小时进行计数。这就允许世界上各个时区跟 UTC 之间相差的时间不是整数个小时。如果这个串为空，那么时间函数就从环境变量 "TIMEZONE" 中寻找一个代替字符串。（可以为 dst_rules 添加一个相似的代替者。）如果也找不到这个环境变量，那么函数就会去寻找一个使用更广泛的环境变量 "TZ"。那个串具有 EST05EDT 这样的形式，这里中间的数字记录 UTC 西部的小时数。

串 dst_rules 甚至更夸张。它使用下面两种一般格式中的一种：

```
(YYYY)MMDDHH+W
(YYYY)MMDDHH-W
```

夏令时

这里，圆括号中的 YYYY 是年，MM 是月，DD 是这个月中的第几天，W 是指星期几，HH 是 24 小时制的天中第几小时。+W 提前到讨论中年份的日期 MMDD 当前或者之后的下一个星期中的这一天，-W 倒退到指定日期之前的上一个星期中的这一天。用来指定年、小时和星期的域可以省略。

上面的那个简单的例子要求夏令时从 3 月 24 日（MMDD=0324）02:00（HH=02）开始，到 10 月 27 日相同的时间结束。为了选择 1990 年之后每年的 3 月和 10 月的最后一个星期天，可以这样写：(1990)040102-0:100102-0。（在这个规则集下，1990 年以前的年份不使用夏令时。）

在南半球，一年刚开始就处于夏令时中。通过再添加第三个反转的域，像 :0101: 030202 :100202 这样，就能体会到其中的差别了。也可以编写任意数目的倒退时间的年份规则。用一个起始年份（YYYY）来限制每一个集合中的第一个规则，来使规则生效。如果你愿意，就可以获得一个特定的州或者地区中由法律控制的夏令时的全部历史。

<ctype.h> 中声明的所有函数都是围绕着转换表组织的。（参考第 2 章）每一个表都是一个具有 257 个 short 类型的元素的数组，这个数组的下标在 [-1,255] 范围内。在区域设置文件中，下标为 -1 的元素的内容是不能改变的，它定义了 <stdio.h> 中定义的宏 EOF 的值。这些表的实体名字是：

```
ctype
tolower
toupper
```

可以对这些表进行初始化，一次初始化一个元素或者一个子集。例如，下面是使用 ASCII 字符和瑞典语的 'Å' 对 tolower 表进行的完整说明：

```
tolower[0 : 255]        $@
tolower['A' : 'Z']      $$ + 'a' - 'A'
tolower['Å' ]           'å'
```

其中特殊项 $@ 是指子集中每一个元素的索引值。（可以把这个项读做“它在哪儿”。）特殊项 $$ 是指表元素的以前的内容的值。（读做“它的值是什么”）。注意可以用一个简单（单字符）的字符常量来指定它的编码值，并且可以对一系列项进行加减操作。当然，前面的两行是可选的。可以从 "C" 区域设置那里继承而得到它们。

状态表

就像本节前面所讨论的那样，这个实现中有几对函数使用状态表来定义它们的行为。对下面每个实体名最多可以指定 16 个状态表。

```
collate
mbtowc
wctomb
```

我会结合使用这些表的函数，更加详细的讲解它们。现在，我只给出一个简单的例子。它显示了怎样为文件 xstate.c（参考图 6-11）中的状态表编写规格说明。它让 <stdlib.h> 中声明的函数 mbtowc 和 mbstowcs 实现了多字节字符和宽字节字符之间一到一的映射。

```
mb_cur_max              1
mbtowc[0, 0:$#]         $@ $F $I $O $0
```

第一行给出了 <stdlib.h> 中定义的宏 MB_CUR_MAX 的值为 1。多字节序列要求不超过一个字符。第二行把 mbtowc 和 mbstowcs 的状态表的所有元素定义为零。它告诉函数：

- 把翻译的值加到累加值（$F）中；
- 让输入编码和它本身相对应（$@）；
- 使用输入（$I）；
- 把累加值作为输出（$O）写出。

后继的状态是状态零（$0）。在这种情况下，当一个零输入编码生成一个零宽字节字符时，翻译就结束了。

表达式 那就是可以在一个区域设置中指定的实体的列表。现在你就能理解为什么某些奇怪的项会出现在表达式中。一个表达式本身就是一个简单地将项加在一起而构成的序列，上面的最后一个例子说明了可以通过简单地把一个项写到另一个后面来把它添加进去。在一个项前面添加一个加号也是可以接受的，它只是提高了表达式的可读性。

项 可以编写很多不同的项。

- 十进制、八进制和十六进制数遵守 C 常量的通用规则。序列 10、012 和 0xA 都表示了十进制的值 10。
- 一个项前的加号不改变它的值，一个减号使它变为负值。
- 一个字符周围的单引号表示该字符的值，就像 C 源文件中的字符常量一样。（没有转义序列，尽管如此，像 '\012' 这样也是允许的。）
- 一个大写字母的值是 SET 最后一次赋予它的值。所有这样的变量在程序启动的时候都设置为零。

$x 项 除了这些项，以一个美元符号开头的双字符名字都有特殊的含义，就像下面列出来的那样。下面就是用一个美元符号说明的特殊项。

- $$——一个表元素的当前内容。
- $@——一个表元素的索引。$$ 和 $@ 如果在一个表达式中出现的话，一定要放在其他的项之前。
- $^——宏 CHAR_MAX 的值。
- $#——宏 UCHAR_MAX 的值。
- [$a $b $f $n $r $t $v]——字符转义序列的值，按顺序依次是['\a' '\b' '\f' '\n' '\r' '\t' '\v']。
- [$A $C $D $H $L $M $P $S $U $W]——表 ctype 中使用的字符分类位，它们依次说明了：额外的字母、额外的控制字符、数字、十六进制数字、小写字母、运动控制字符、标点符号、空白字符、大写字母和额外的空白字符。（参考 2.4 节对相关的宏的定义。）
- [$0 $1 $2 $3 $4 $5 $6 $7]——一个状态表元素中状态 0 到 7 的后继状态。（对后继状态 8 到 15 不提供符号。对状态 8，可以写作 $7+$1，依次类推。）
- [$F $I $O $R]——状态表元素中使用的命令位。这些项按照顺序说明了：把翻译值加到累加值中，使用输入，产生输出和累加值中的翻转字节。（参考图 6-5 的文件 xstate.h 中相关宏的定义。）

通过这些特殊的项，就可以在区域设置文件中编写表达式，而不依赖于实现指定的编码值。

"USA" 区域设置

我用一个完整的区域设置的例子来结束这一部分。下面是 "USA" 区域设置，它对 struct lconv 中的所有的域都赋了合理的值。对 "C" 区域设置中的排列顺序和多字节编码没有作任何修改：

```
LOCALE                 USA
currency_symbol        "$"
decimal_point          "."
grouping               "3"
int_curr_symbol        "USD"
mon_decimal_point      "."
mon_grouping           "3"
mon_thousands_sep      ","
negative_sign          "-"
positive_sign          "+"
thousands_sep          ","
frac_digits            2
int_frac_digits        2
n_cs_precedes          1
n_sep_by_space         0
n_sign_posn            4
p_cs_precedes          1
p_sep_by_space         0
p_sign_posn            4
LOCALE                 end
```

最后一行界定了区域设置的结束。只有在区域设置文件中最后一个区域设置的结尾才需要这样的一行（但是它总是允许的）。为了改进检查体制，如果一个区域设置的中间部分出现了文件结束符，那些读入区域设置文件的函数就会报告一个错误。

函数 _Getloc 的二次访问

现在就可以理解实现 <locale.h> 的剩下的函数了。还记得吗？_Getloc（图 6-8）首先尝试在内存中找到一个区域设置。如果失败了，它就会尝试打开区域设置文件，并从中扫描到需要的区域设置的起始位置。它只查找区域设置文件中以关键字 LOCALE 开头的行。_Getloc 调用 _Readloc 来读取每一行并且识别它的关键字。

如果 _Getloc 找到这个关键字后面跟着它要找的名字的所在行，这个函数就为新的区域设置分配空间。它首先复制 _Clocale 的内容，然后改变为新的名字。函数 _Makeloc 为区域设置读取剩下的信息，并且修改相应的存储区域。如果 _Makeloc 报告成功，_Getloc 就把新的区域设置添加到起始位置为 _Clocale._Next 的列表上。如果 _Makeloc 报告失败，_Getloc 向标准错误流输出一条错误信息，放弃所有分配的空间，并且报告它不能找到该区域设置。错误信息中的一部分是导致失败的区域设置文件中的那一行。

作为一个规则，库函数输出这样的错误信息不是很好的做法。它们抢占了程序员决定怎样最好地从一个错误中恢复的权利。然而我发现，在这种情况下这种信息是非常有价值的。如果 setlocale 只读取一个畸形的区域设置的一部分或者隐秘地拒绝读取，那么该区域设置就很难调试。库已经纵容了

一个涉及打开和读取文件的复杂操作，而且还可能重复地执行——所有的都是为了关照程序员，让它看起来像一个简单的函数调用。从这个角度看，输出到标准错误流不是一个主要的附加物。（在某些环境下，可能仍然需要选择省略输出。）

函数 _Readloc

图 6-12 显示了文件 xreadloc.c。它定义了函数 _Readloc，这个函数读取区域设置文件，一次读取一行。调用者提供一个 MAXLIN 长度的 buf 缓冲区来放这一行内容。（头文件 "xlocale.h" 把宏 MAXLIN 定义为 256。）这就是以反斜线结尾的一行和下一行连接起来的地方，也是关键字被分析，识别，并且从每一行的开头去掉的地方。

_Readloc 使用表达式 (n=strspn(s,kc)) 来确定输入一行内容中的关键字的长度。这个表达式将字符串 kc 中从 s 处开始的最长的字符序列的长度

图 6-12 xreadloc.c

```
/* _Readloc function */
# include <stdio.h>
# include <string.h>
# include "xlocale.h"

        /* static data */
static const char kc[] =                          /* keyword chars */
    "_abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";
_Locitem *_Readloc(FILE *lf, char *buf, const char **ps)
   {                                    /* get a line from locale file */
    for (; ; )
        {                               /* loop until EOF or full line */
        size_t n;

        for (buf[0] = ' ', n = 1 ; ; n-=2)
            if (fgets(buf + n, MAXLIN - n, lf) == NULL
                || buf[(n += strlen(buf + n)) - 1] != '\n')
                return (NULL);              /* EOF or line too long */
            else if (n <= 1 || buf[n - 2] != '\\')
                break ;                 /* continue only if ends in \ */
        buf [n - 1] = '\0';                       /* overwrite newline */
         {                                 /* look for keyword on line */
        const char *s = _Skip(buf);
        _Locitem *q;

        if (0 < (n = strspn(s, kc)))
            for (q = _Loctab; q->_Name; ++q)
                if (strncmp(q->_Name, s, n) == 0
                    && strlen (q->_Name) == n)
                    {                                 /* found a match */
                    *ps = _Skip(s + n - 1);
                    return (q);
                    }
        return (NULL);                 /* unknown or missing keyword */
         }
        }
    }
```

存储在 n 中。我没有使用 <ctype.h> 中像 isalpha 这样的字符分类函数，因为它们会随着区域设置不同而改变。

类型 _Locitem

_Readloc 在 *ps 处存储了一个指针，该指针指向关键字后面或者任何空白后面的那行的第一个字符。该函数也返回一个指针，该指针指向一个包含它识别的关键字信息的表项。头文件 "xlocale.c" 把类型 _Lcode 和 _Locitem 定义为：

```
enum _Lcode {
    L_GSTRING, L_NAME, L_NOTE, L_SET,
    L_STATE, L_STRING, L_TABLE, L_VALUE
    };
typedef struct {
    const char *_Name;
    size_t _Offset;
    enum _Lcode _Code;
    } _Locitem;
```

（标量类型 size_t 是操作符 sizeof 产生的整数类型。有几个标准头文件都定义了这个类型。我会在第 11 章详细地讨论它。）成员 _Name 指向关键字的名字。_Offset 将偏移量保存到和关键字（如果有的话）对应的成员的结构 _Linfo 中。_Code 用来保存一个描述 _Locitem 的每一个实例的特征的枚举值。

数据对象 _Loctab

_Readloc 对数据对象 _Loctab 和数组 _Locitem 进行扫描，以寻找和区域设置文件中每一行的关键字匹配的入口。图 6-13 显示了文件 xloctab.c，它定义了 _Loctab。该文件使用 <stddef.h> 中定义的宏 offsetof 来确定和结构 _Linfo 的偏移量。这里我使用宏 OFF 来缩短这个 C 源文件中的文本行。

函数 _Freeloc

还有一个函数使用了 _Loctab。图 6-14 显示了文件 xfreeloc.c。它定义了函数 _Freeloc。如果 _Makeloc 在读取区域设置文件时遇到了一个无效行，它就回来向 _Getloc 报告失败。_Getloc 函数就调用 _Freeloc 来释放为新的区域设置分配的所有空间（包括它的名字），然后释放为新的区域设置分配的 _Linfo 数据对象。（放弃这样的空间可能也是可以接受的——要求一个有缺陷的区域设置很少出现——但是收回不再使用的堆空间会变得更加整洁。）_Freeloc 扫描 _Loctab，寻找那些与 _Linfo 的成员相对应的元素，这些成员可以通过在区域设置文件中写入文本行而被修改。对 _Loctab 的每一个这样的元素，_Freeloc 确定是否为新的区域设置分配了存储空间。实现这些要花费点精力。

记住，每一个新的区域设置开始都是 "C" 区域设置的一个副本。_Makeloc 只有当一个区域设置文件行需要改变的时候才分配一个新的表或者串。请求这样的一个改变，然后 _Makeloc 对新的 _Linfo 数据对象的相关的指针成员和 _Clocale 进行比较。如果这些指针相同，_Makeloc 就知道分配一个新的版本。改变也适用于新的版本，而把 "C" 区域设置的数据放到一边。如果指针不同，_Makeloc 就会假设它已经为这个新的区域设置分配了一个新的

版本。新版本中的改动会逐渐增多。

_Freeloc 执行相似的测试。如果它遇到一个指向串或者表的指针，且该指针与 _Clocale 中相对应的指针匹配，那么该指针保持不变。如果它遇到一个指针，且该指针在新的区域设置和 _Clocale 之间不同，那么它就释放新的存储空间。

图 6-13
xloctab.c

```
/* _Loctab data object */
#include <stddef.h>
#include "xlocale.h"

        /* macros */
#define OFF(member)     offsetof(_Linfo, member)
        /* static data */
_Locitem _Loctab[] = {                          /* locale file info */
    "LOCALE", OFF(_Name), L_NAME,
    "NOTE", 0, L_NOTE,
    "SET", 0, L_SET,
        /* controlled by LC_COLLATE */
    "collate", OFF(_Costate._Tab), L_STATE,
        /* controlled by LC_CTYPE */
    "ctype" OFF(_Ctype), L_TABLE,
    "tolower", OFF(_Tolower), L_TABLE,
    "toupper", OFF(_Toupper), L_TABLE,
    "mb_cur_max", OFF(_Mbcurmax), L_VALUE,
    "mbtowc", OFF(_Mbstate._Tab), L_STATE,
    "wctomb", OFF(_Wcstate._Tab), L_STATE,
        /* controlled by LC_MONETARY */
    "currency_symbol", OFF(_Lc.currency_symbol), L_STRING,
    "int_curr_symbol", OFF(_Lc.int_curr_symbol), L_STRING,
    "mon_decimal_point", OFF(_Lc.mon_decimal point), L_STRING,
    "mon_grouping", OFF(_Lc.mon_grouping), L_GSTRING,
    "mon_thousands_sep", OFF(_Lc.mon_thousands_sep), L_STRING,
    "negative_sign", OFF(_Lc.negative_sign), L_STRING,
    "positive_sign", OFF(_Lc.positive_sign), L_STRING,
    "frac_digits" OFF(_Lc.frac_digits), L_VALUE,
    "int frac_digits" OFF(_Lc.int_frac_digits), L_VALUE,
    "n_cs_precedes" OFF(_Lc.n_cs_precedes), L_VALUE,
    "n_sep_by_space", OFF(_Lc.n_sep_by_space), L_VALUE,
    "n_sign_posn", OFF(_Lc.n_sign_posn), L_VALUE,
    "p_cs_precedes" OFF(_Lc.p_cs_precedes), L_VALUE,
    "p_sep_by_space", OFF(_Lc.p_sep_by_space), L_VALUE,
    "p_sign_posn", OFF(_Lc.p_sign_posn), L_VALUE,
        /* controlled by LC_NUMERIC */
    "decimal_point", OFF(_Lc.decimal_point), L_STRING,
    "grouping", OFF(_Lc.grouping), L_GSTRING,
    "thousands_sep", OFF(_Lc.thousands_sep), L_STRING,
        /* controlled by LC_TIME */
    "am_pm", OFF(_Times._Ampm), L_STRING,
    "days" OFF(_Times._Days), L_STRING,
    "dst_rules", OFF(_Times._Isdst), L_STRING,
    "time_formats", OFF(_Times._Formats), L_STRING,
    "months", OFF(_Times._Months), L_STRING,
    "time_zone", OFF(_Times._Tzone), L_STRING,
    NULL};
```
□

图 6-14
xfreeloc.c

```
/* _Freeloc function */
#include "xlocale.h"

void _Freeloc(_Linfo *p)
    {                                          /* free all storage */
    _Locitem *q;

    for (q = _Loctab; q->_Name; ++q)
        switch (q->_Code)
            {                                 /* free all pointers */
        case L_STATE:
            {                             /* free all state entries */
            int i;
            unsigned short **pt
                = &ADDR(p, q, unsigned short *);

            for (i = _NSTATE; 0 <= --i; ++pt)
                if (*pt && (*pt)[-1] != 0)
                    free(*pt);
            }
            break;
        case L_TABLE:
            if (NEWADDR(p, q, short *))
                free(ADDR(p, q, short *) - 1);
            break;
        case L_GSTRING:
        case L_NAME:
        case L_STRING:
            if (NEWADDR(p, q, char *))
                free(ADDR(p, q, char *));
            }
    }
```

_Makeloc 和 _Freeloc 都使用两个很复杂的宏来完成这个工作。头文件 "xlocale.h" 包含了以下的定义：

```
#define ADDR(p, q, ty) (*(ty *)((char *)p + q->_Offset))
#define NEWADDR(p, q,ty)    \
    (ADDR(p,q,ty) != ADDR(&_Clocale, q, ty))
```

宏 ADDR 例如，可以编写 ADDR(p,q,char*) 来构建一个左值——一个可以用来访问数据对象的部分或者全部的表达式。这里，数据对象是由 p 指向的 _Linfo 类型的结构的一个成员。q 指向 _Loctab（_Locitem 类型）的一个元素，这个元素包含了这个成员的偏移量。这种情况下，这个成员为指向 *char* 类型数据的指针。

宏 NEWADDR 例如，可以写 NEWADDR(p, q, char *) 来测试一个成员从 _Clocale 复制出来以后是否发生了改变。参数和宏 ADDR 的相同。

释放状态表 然而，这种机制为状态表打破了。每一个状态表都包含了指向可以在区域设置文件中指定的表的 _NSTATE 个指针。（头文件 "xstate.h" 中把宏 _NSTATE 定义为 16。）正如它们所表示的，这些宏对每一个需要条件释放的表都需要 _Loctab 中的一个元素。我不想向 _Loctab 灌输很多虚拟的项来考

验 _Freeloc 的能力。同样，我也不想让宏 ADDR 和 NEWADDR 更加复杂。

因此，我决定对释放状态表使用一种不同的机制。为了分享编码，我已经将状态表构造得看起来很像 <ctype.h> 中声明的函数使用的字符转换表。那就是说，每一个表都有一个下标为 -1（和 <stdio.h> 中定义的宏 EOF 的值相对应）的元素。那些使用状态表的函数中没有一个知道或者关心这个额外的元素。所以，我把它当作一个表示状态表是否已经分配的标志。

"C" 区域设置中的所有函数共享的原始的状态表在文件 xstate.c（参考图 6-11）中定义。它的下标为 -1 的元素值为零。如果 _Makeloc 分配一个新的状态表，它就在下标为 -1 的元素中存储一个非零值。这就是 _Freeloc 知道是否要释放一个状态表的方式。

函数 _Makeloc 图 6-15 显示了文件 xmakeloc.c。它定义了函数 _Makeloc（已经详细讨论了这个函数）。虽然 _Makeloc 很大，但它只是一个处理区域设置文件中的文本行的简单的 *while* 循环。循环的主体部分是处理不同种类的文本行的 *switch* 语句。这些代码很简单，但是冗长而又紧凑。

6

还没有遇到过的一个宏是 TABSIZ。头文件 "xlocale.h" 包含了下面的定义：

```
#define TABSIZ ((UCHAR_MAX + 2) * sizeof (short))
```

这是一种简单的、书写各种各样的表的大小的可移植的方式，这些表的大小以字节为单位，可以在一个区域设置文件中修改它们。

_Makeloc 尽可能多地调用内部函数 getval 来对表达式进行分析和求值。这就有助于在一个区域设置文件行中写入表达式时保持规则的统一。（表元素的表达式是一个例外——只有它们接受特殊项 $@ 和 $$。）反过来，getval 重复地调用 _Locterm 来计算一个项的序列的和。

函数 _Locterm 图 6-16 显示了文件 xlocterm.c，这个文件定义了函数 _Locterm。这就是各个项被分析和求值的地方。为了求八进制、十进制和十六进制的数值，_Locterm 调用 <stdlib.h> 中声明的 strtol。注意这个函数是如何更新字符指针 s 来指向它刚刚分析和转换的数字。_Locterm 的代码被极大地压缩了。

函数 _Locvar 文件 xlocterm.c 也定义了函数 _Locvar。只有 _Makeloc 用 SET 关键字处理一个区域设置文件文本行的时候才调用该函数。_Locvar 也很小，它可以简单地用内联代码来取代。

然而，我把 _Locvar 放到 xlocterm.c 中却有一个很好的理由。它和 _Locterm 共享了访问两个数组 uppers 和 vars 的需求。这就分别给出了可以在区域设置文件中用 SET 关键字修改的项的名字和值。通过把这两个函数放到同一个文件中，这些数组可以对那个文件保持私有，就像它们的实现细节可以做的那样。

图 6-15
xmakeloc.c

```
/* _Makeloc function */
#include <string.h>
#include "xlocale.h"

static const char *getval(const char *s, unsigned short *ans)
    {                                          /* accumulate terms */
    unsigned short val;

    if (!_Locterm(&s, ans))
        return (NULL);
    while (_Locterm(&s, &val))
        *ans += val;
    return (s);
    }

int _Makeloc(FILE *lf, char *buf, _Linfo *p)
    {                               /* construct locale from text file */
    const char *s;
    char *s1;
    _Locitem *q;
    unsigned short val;
    static const char gmap[] = "0123456789abcdef^";

    while ((q = _Readloc(lf, buf, &s)) != NULL)
        switch (q->_Code)
            {                                       /* process a line */
        case L_GSTRING:                      /* alter a grouping string */
        case L_STRING:                         /* alter a normal string */
            if (NEWADDR(p, q, char *))
                free(ADDR(p, q, char *));
            if (s[0] == '"'
                && (s1 = strrchr(s + 1, '"')) != NULL
                && *_Skip(s1) == '\0')
                *s1 = '\0', ++s;
            if ((s1 = (char *)malloc(strlen(s) + 1)) == NULL)
                return (0);
            ADDR(p, q, char *) = strcpy(s1, s);
            if (q->_Code == L_GSTRING)
                for (; *s1; ++s1)
                    if ((s = strchr(gmap, *s1)) != NULL)
                        *s1 = *s == '^' ? CHAR_MAX : s-gmap;
            break;
        case L_TABLE:                      /* alter a translation table */
        case L_STATE:                            /* alter a state table */
            {                      /* process tab[#,lo:hi] $x expr */
            int inc = 0;
            unsigned short hi, lo, stno, *usp, **uspp;

            if (*s != '['
                || (s = getval(_Skip(s), &stno)) == NULL)
                return (0);
            if (*s != ',')
                lo = stno, stno = 0;
            else if (q->_Code != L_STATE || _NSTATE <= stno
                || (s = getval(_Skip(s), &lo)) == NULL)
```

图 6-15
(续)

```
            return (0);
        lo = (unsigned char)lo;
        if (*s != ':')
            hi = lo;
        else if ((s = getval(_Skip(s), &hi)) == NULL)
            return (0);
        else
            hi = (unsigned char) hi;
        if (*s != ']')
            return (0);
        for (s = _Skip(s); s[0] == '$'; s = _Skip(s + 1))
            if (s[1] == '@' && (inc & 1) == 0)
                inc |= 1;
            else if (s[1] == '$' && (inc & 2) == 0)
                inc | = 2;
            else
                break;
        if ((s = getval(s, &val)) == NULL || *s != '\0')
            return (0);
        uspp = &ADDR(p, q, unsigned short *) + stno;
        if (q->_Code == L_TABLE)
            usp = NEWADDR(p, q, short *) ? *uspp : NULL;
        else
            usp = (*uspp) [-1] ? *uspp : NULL;
        if (usp == NULL)
            {                                   /* setup a new table */
            if ((usp = (unsigned short *)malloc(TABSIZ))
                == NULL)
                return (0);
            usp[0] = EOF;                   /* allocation flag or EOF */
            memcpy(++usp, ADDR(p, q, short *),
                TABSIZ - sizeof (short));
            *uspp = usp;
            }
        for (; lo <= hi; ++lo)
            usp[lo] = val + (inc & 1 ? lo : 0)
                + (inc & 2 ? usp[lo] : 0);
     }
        break;
    case L_VALUE:                             /* alter a numeric value * /
        if ((s = getval(s, &val)) == NULL || *s != '\0')
            return (0);
        ADDR(p, q, char) = val;
        break;
    case L_SET:                        /* assign to uppercase variable * /
        if (*(s1 = (char *)_Skip(s)) == '\0'
            || (s1 = (char *)getval (s1, &val)) == NULL
            || *s1 != '\0' || _Locvar(*s, val) == 0)
            return (0);
        break;
    case L_NAME:                        /* end happily with next LOCALE */
        return (1);
        }
return (0);                         /* fail on EOF or unknown keyword */
}
```
□

图 6-16
xlocterm.c

```
/* _Locterm and_Locvar functions */
#include <ctype.h>
#include <limits.h>
#include <string.h>
#include "xlocale.h"

        /* static data */
static const char dollars[] = {                    /* PLUS $@ and $$ */
    "^abfnrtv"                                    /* character codes */
    "01234567"                                       /* state values */
    "ACDHLMPSUW"                                      /* ctype codes */
    "#FIOR"};                                      /* state commands */
static const unsigned short dolvals[] = {
    CHAR_MAX, '\a', '\b', '\f', '\n', '\r', '\t', '\v',
    0x000, 0x100, 0x200, 0x300, 0x400, 0x500, 0x600, 0x700,
    _XA, _BB, _DI, _XD, _LO, _CN, _PU, _SP, _UP, _XS,
    UCHAR_MAX, ST_FOLD, ST_INPUT, ST_OUTPUT, ST_ROTATE};
static const char uppers [] = "ABCDEFGHIJKIMNOPQRSTUVWXYZ";
static short vars[sizeof (uppers) - 1] = {0};

int _Locvar(char ch, short val)
    {                                            /* set a $ variable */
    const char *s = strchr(uppers, ch);

    if (s == NULL)
        return (0);
    vars[s - uppers] = val;
    return (1);
    }

int _Locterm(const char **ps, unsigned short *ans)
    {                         /* evaluate a term on a locale file line */
    const char *s = *ps;
    const char *s1;
    int mi;

    for (mi = 0; *s = '+' || *s == '-' ; s = _Skip(s))
        mi = *s == '_' ? !mi : mi;
    if (isdigit(s[0]))
        *ans = strtol(s, (char **) &s, 0);
    else if (s[0] == '\'' && s [1] != '\0' && s[2] == '\'')
        *ans = ((unsigned char *)s) [1], s += 3;
    else if (s[0] && (s1 = strchr(uppers, s[0])) != NULL)
        *ans = vars[s1 - uppers], ++s;
    else if (s[0] == '$' && s[1]
        && (s1 = strchr(dollars, s[1])) != NULL)
        *ans = dolvals[s1 - dollars], s += 2;
    else
        return (0);
    if (mi)
        *ans = -*ans;
    *ps = _Skip(s - 1);
    return (1);
    }
```

□

头文件 "xlocale.h"

最后，我将给出内部头文件 "xlocale.h" 的完整内容。图 6-17 显示了文件 xlocale.h。到现在再显示这个文件就有点平淡了。因为在前面的过程中你已经看到了所有重要的部分了。

事实上，本章出现了大约 800 行的代码。对于实现 C 标准库中描述的仅有的两个函数和一个标准头文件来说，这些代码太多了。然而，我相信，定义新的区域设置的能力提供了相当多的承诺。如果对代码的投入能传递到那些承诺身上，它们的价值就体现出来了。

6.5 <locale.h> 的测试

图 6-18 显示了测试程序 tlocale.c。它主要关注 <locale.h> 中的函数的可移植的行为。因此，对本章中出现的很多代码它都没有测试。要做到这一点，你需要转换到一个新的区域设置，例如前面给出的"USA"。然后可以打印附加函数 _Fmtval 的结果来检验行为是否按照预想的改变了。

可以使用 tlocale.c 来测试标准 C 的任何实现。它保证 "C" 区域设置满足 C 标准的所有要求，无论是在区域设置的各种改变之前还是之后。它还保证可以建立混合的区域设置——至少包含了 "C" 区域设置和本地区域设置。它尽力确定这两个区域设置是否不同。你会得到这两条信息之一。这个实现下应该输出：

```
Native locale same as "C" locale
SUCCESS testing <locale.h>
```

6.6 参考文献

ISO Standard 4217:1987 (Geneva: International Standards Organization, 1987). 这个标准对各个国家的货币的三字母编码进行了详细说明。

6.7 习题

6.1 编写表示意大利、荷兰、挪威和瑞士的货币习惯的区域设置。使用 C 标准 7.4.2.1 部分的例子提供的信息（见 6.2 节）。

6.2 编写一个表示法语的字符分类习惯的区域设置。把小写字母 [á à â ç é è ê ô û] 和它们相应的大写字母 [Á À Â Ç É È Ê Ô Û] 添加到转换表 ctpye、tolower 和 toupper 中。你怎样在你的实现下确定这些字母的编码值？

图 6-17
xlocale.h

```
/* xlocale.h internal header */
#include <limits.h>
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>
#include "xstate.h"
#include "xtinfo.h"

        /* macros for _Getloc and friends */
#define ADDR(p, q, ty)(*(ty *)((char *)p + q->_Offset))
#define NEWADDR(p, q, ty) \
    (ADDR(p, q, ty) != ADDR(&_Clocale, q, ty))
#define MAXLIN 256
#define TABSIZ ((UCHAR_MAX + 2) * sizeof (short))
        /* type definitions */
typedef const struct {
    const char *_Name;
    size_t _Offset;
    enum {
        L_GSTRING, L_NAME, L_NOTE, L_SET,
        L_STATE, L_STRING, L_TABLE, L_VALUE
        } _Code;
    } _Locitem;
typedef struct _Linfo {
    const char * _Name;                          /* must be first */
    struct _Linfo *_Next;
        /* controlled by LC_COLLATE */
    _Statab _Costate;
        /* controlled by LC_CTYPE */
    const short *_Ctype;
    const short *_Tolower;
    const short *_Toupper;
    unsigned char _Mbcurmax;
    _Statab _Mbstate;
    _Statab _Wcstate;
        /* controlled by LC_MONETARY and LC_NUMERIC */
    struct lconv _Lc;
        /* controlled by LC_TIME */
    _Tinfo _Times;
    } _Linfo;
        /* declarations */
const char *_Defloc(void);
void _Freeloc(_Linfo *);
_Linfo *_Getloc(const char *, const char *);
int _Locterm(const char **, unsigned short *);
int _Locvar(char, short);
int _Makeloc(FILE *, char *, _Linfo *);
_Locitem *_Readloc(FILE *, char *, const char **);
_Linfo *_Setloc(int, _Linfo *);
const char *_Skip(const char *);
extern _Linfo _Clocale;
extern _Locitem _Loctab[];
```
□

图 6-18
tlocale.c

```
/* test locales */
#include <assert.h>
#include <limits.h>
#include <locale.h>
#include <stdio.h>
#include <string.h>

static void testclocale(struct lconv *p)
    {                                 /* test properties of "C" locale */
    assert(strcmp(p->currency_symbol, "") == 0);
    assert(strcmp(p->decimal_point, ".") == 0);
    assert(strcmp(p->grouping, "") == 0);
    assert(strcmp(p->int_curr_symbol, "") == 0);
    assert(strcmp(p->mon_decimal_point, "") == 0);
    assert(strcmp(p->mon_grouping, "") == 0);
    assert(strcmp(p->mon_thousands_sep, "") == 0);
    assert(strcmp(p->negative_sign,"") == 0);
    assert(strcmp(p->positive_sign,"") == 0);
    assert(strcmp(p->thousands_sep, "") == 0);
    assert(p->frac_digits == CHAR_MAX);
    assert(p->int_frac_digits == CHAR_MAX);
    assert(p->n_cs_precedes == CHAR_MAX);
    assert(p->n_sep_by_space == CHAR_MAX);
    assert(p->n_sign_posn == CHAR_MAX);
    assert(p->p_cs_precedes == CHAR_MAX);
    assert(p->p_sep_by_space == CHAR_MAX);
    assert(p->p_sign_posn == CHAR_MAX);
    }

int main()
    {                              /* test basic properties of locales */
    static int cats [] = {LC_ALL, LC_COLLATE, LC_CTYPE,
        LC_MONETARY, LC_NUMERIC, LC_TIME } ;
    struct lconv *p = NULL;
    char buf[32], *s;

    assert((p = localeconv()) != NULL);
    testclocale(p);
    assert((s = setlocale(LC_ALL, NULL)) ! = NULL);
    assert(strlen(s) < sizeof (buf));                 /* OK if longer */
    strcpy(buf, s);               /* but not safe for this program */
    assert(setlocale(LC_ALL, "") != NULL);
    assert(localeconv() != NULL);
    assert((s = setlocale(LC_MONETARY, "C")) != NULL);
    puts(strcmp(s, "C") ? "Native locale differs from \"C\""
        : "Native locale same as \"C\"");
    assert(setlocale(LC_NUMERIC, "C") != NULL);
    assert((p = localeconv()) != NULL);
    testclocale(p);
    assert(setlocale(LC_ALL, buf) != NULL);
    assert((p = localeconv()) != NULL);
    testclocale(p);
    puts("SUCCESS testing <locale.h>");
    return (0);
    }
```

□

6.3 修改测试程序 tctype.c（图 2-19），使它首先转换为上题中的区域设置。当你运行它的时候，它会显示你所希望的内容吗？

6.4 编写表示法语的币值和数字习惯的区域设置，你至少需要修改以下几个地方：

```
mon_decimal_point   decimal_point
mon_grouping        grouping
mon_thousands_sep   thousands_sep
negative_sign       positive_sign
```

测试你的新的区域设置。（提示：你可以从本章和后面的章节中的测试程序入手。）

6.5 [难] 有很多具有小数位的值的表经常以 5 为单位对小数点右边的数字进行分组。例如：

```
+1.00000 00000 00
-0.16666 66666 67
+0.00833 33333 33
-0.00019 84126 98
```

向 struct lconv 中添加成员 frat_grouping 和 frat_group_sep。用这样的方式定义它们：你可以指定这个例子（当然也包括其他的）中使用的格式。修改本章中的代码，包括 _Fmtval，来对这些成员正确地进行初始化、复制、修改和使用。C 标准允许这样的增加吗？

6.6 [难] 你希望有一个可以构建它自己的区域设置的程序。不能重新编写区域设置文件。你会在 <locale.h> 中添加什么函数来允许一个程序迅速地命名、构建和添加新的区域设置。编写一个用户文档，程序员能用这个文档来增加区域设置。

6.7 [很难] 实现你在上题中描述的功能。

第7章

<math.h>

7

7.1 背景知识

编写出好的数学函数是一件很难的事。现在的一个普遍现象是有些程序设计语言的实现提供的数学函数存在严重的缺陷。它们可能会对具有明确定义的函数值的参数产生中间值溢出，或者丢失很多位的有效数字，某些情况下还会产生错误结果。

历史 虽然实现人员有充足的时间来研究这些问题，但仍然有那么多的缺陷，这一点让人奇怪。计算机最早就用于解决各种工程或者数学难题，事实上，最早的库几乎完全由计算常用数学函数的函数组成。20 世纪 50 年代出现的 FORTRAN，就是以它的功能 FORmula TRANslation（公式转换）的缩写来命名的，那些公式是由很多数学函数组成的。

在过去的几十年中，实现人员变得越来越老练。IEEE 754 浮点数标准对浮点算术的安全和一致是一个重要的里程碑。（参考第 4 章中对浮点数表示和 IEEE 754 标准的讨论。）但是从另一个角度来说，IEEE 754 又加重了实现人员的负担。因为它引入了渐进下溢，对无穷大和非数的编码，不同的精度要求不同大小的指数等复杂的东西。所以很多实现通常只支持 IEEE 754 标准的一部分，而不是全部。

我花在编写和调试 `<math.h>` 中声明的函数的时间和这个库中其他所有的函数加起来的时间差不多，这的确让我有点吃惊。在过去的二十多年的时间中，我对每一个数学库函数都至少预先编写过 3 遍所以你可能会认为我应该有足够的时间来避免错误的发生，我也曾这样认为，但事实并非如此。

目标 我花了这么长的时间是为了达到以下几个目标：

- 数学库函数应该能在各种常用的计算机体系结构之间移植。每个函数产生的结果都要具有 56 位的精度，这会使它们适合很多 64 位 *double* 表示的机器——那些具有 IEEE 754 可兼容的协处理器的机器（53 位精度）、IBM System/370 系列（53 ～ 56 位）和 DEC VAX 系列（56

位）。

- 每一个函数都应该能接受定义域内的所有参数值（即数学上定义的参数值）。对于其他所有的参数都应该报告定义域错误，这种情况下，函数返回一个表示非数字的特殊编码 NaN。
- 对于一个具体的值，函数应该产生一个确定的结果。对于所有太大或者太小而不能表示的值，它应该报告值域错误。如果结果的数值太大，那么函数就要返回一个表示正无穷大的特殊编码 +Inf，或者表示负无穷大的编码 -Inf；如果数值太小，函数就返回零。
- 对于参数值 NaN、+Inf 和 -Inf，每一个函数都应该产生最合理的结果。在一个像 IEEE 754 这样支持多重 NaN 编码的实现下，这些函数在任何需要的地方都要保留特殊的 NaN 编码。例如，如果一个函数只有一个参数并且那个参数的值为 NaN，那么这个函数就返回参数值。
- 对任何可以表示的结果，每个函数都应尽可能地把精度误差控制在两位之内。
- 不论一个函数的参数值或者结果是什么，它都不应该产生上溢、下溢或者除零的情况。
- 每一个函数一般都用双精度数，而不是浮点数来表示中间结果。

本书的代码可以说已经达到了这些目标，因为这些函数从测试开始到现在一直经得起考验。

没有实现的目标

下面是一些没有实现的目标。

- 这个库不区分 +0 和 -0，而 IEEE 754 就严格区分零的这两种形式。我上面提到的所有的体系结构都可以把 0 表示为这两种形式。但是我却不能接受（甚至是理解）其中的道理，近期就有很多人对 IEEE 754 标准提出了批评，并对区分零的原理提出了质疑。在多数情况下，如果一定要处理零的符号，这些函数会很难编写。
- 这个库对各种各样的 NaN 没有做任何区别。例如，IEEE 754 算术就区分静默 NaN（quiet NaN）和信号 NaN（signalling NaN），后者应该产生一个信号或者引发一个异常，但这个实现实际上把所有的 NaN 都作为静默 NaN 处理。
- 我为 IEEE 754 表示提供了低级原语（primitive）。它们恰巧也能对 DEC VAX 的浮点数表示正常地运行，但是并不能完全匹配。VAX 的硬件不能把像 +Inf 和 -Inf 之类的东西识别为特殊的代码值，这些代码会在与它们进行运算的表达式中消失。这些原语一定要经过修改才能支持 System/370 浮点。

- 我没有在 System/370 下检查这些函数，因为那种结构下的“摇摆精度（wobbling precision）”要求特殊的处理。虽然我也大致提供了这种特殊的处理，但它可能不是十分彻底。
- 有些机器的 *double* 类型的保留精度远少于 53 位，对于这些机器，很多函数可能都不是最理想的。C 标准允许 *double* 保留低至 10 位小数的精度——大约 31 位。对这样的机器，应该重新考虑各种数学函数的近似值选择。
- 有些机器的精度多于 56 位。在这些机器下使用近似值的函数几乎都会出错。因此，对这样的机器，也必须考虑重新选择近似值。
- 数学库函数的这个实现不能很好地支持基数不是 2 或者 16 的浮点数表示。例如，一个基数为 10 的浮点数算术的实现，就会要求重新设计有效值。

即使没有达到以上目标，这些库函数的实现仍然适用于各种各样的环境。

安全并且精确地计算数学函数要求下面这些特殊的编程风格：

有限精度

- 浮点数表示的有限精度既有利又有弊。近似值的精度是有限的。但是中间数的计算精度也是有限的。而实际上，这样有时会产生很大的误差。

有限范围

- 浮点数表示的有限的范围也是既有利又有弊。这样可以选择安全的数据类型来表示任意的指数，但又会在中间数的计算中产生上溢或下溢。

7

你可以通过执行各种各样的半数值操作来分解浮点值。可以把浮点值分为以下独立的部分：小范围值的小数、整数指数和符号位。这样你就能以更快的速度、更高的精度和更高的安全性对各个部分进行操作。然后再使用其他的半数字操作把最后的结果连在一起。

Cody 和 Waite

由 William J. Cody、Jr. 和 William Waite 写的 *Software Manual for the Elementary Functions* 是一本关于编写数学函数库的好书。本章中的很多函数都使用了 Cody 和 Waite 描述的算法和技术。很多人实际使用的近似值都是 Cody 和 Waite 特别为他们的这本书设计的。我原以为在某些场合下可以去掉一些他们推荐的繁琐的步骤，但事实证明我是错的。我很庆幸我借鉴了这些严谨的先驱们的成果。

elefunt 测试

最后一点，<math.h> 中声明的很多函数进行的测试是公共域（public_domain）的 `elefunt`（“elementary function”（初等函数）的缩写）测试。这些都源于 Cody 和 Waite 精心设计的测试。

7.2 C 标准的内容

<math.h>

7.5 数学函数 <math.h>

头文件 <math.h> 声明了一些数学函数并定义了一个宏。这些函数接收 double 类型的参数，并返回一个 double 类型值 [103]。以后会讨论整数算术函数和转换函数。

HUGE_VAL

定义的宏是：

```
HUGE_VAL
```

它展开为一个正的 double 类型的表达式，用 float 类型不一定能表达 [104]。

参见：整数算术函数（7.10.6）、函数 atof（7.10.1.1）和函数 strtod（7.10.1.4）。

7.5.1 错误情况的处理

这些函数的行为对输入参数的所有可表示的值都是有定义的。每一个函数执行时都像一个单一操作，而且不能产生任何外部可见的异常。

定义域错误

对所有的函数，如果输入参数位于函数的定义域之外，则发生*定义域错误*。每一个函数的说明都要列出所有的定义域错误。如果一个定义域错误和函数的定义 [105] 一致，那么实现也可以额外地定义这个错误。发生定义域错误时，函数返回一个由实现定义的值，errno 设置为宏 EDOM。

值域错误

相似地，如果结果不能表示为 double 值，则发生*值域错误*。如果结果上溢（结果的数值太大而不能在指定类型的对象中表示），函数返回宏 HUGE_VAL 的值，符号与函数的正确值相同（除了函数 tan），errno 设置为宏 ERANGE。如果结果下溢（结果的数值太小而不能在指定类型的对象中表示），函数返回零，errno 是否取宏 ERANGE 的值是实现定义的。

7.5.2 三角函数

acos

7.5.2.1 函数 acos

概述

```
#include <math.h>
double acos(double x);
```

说明

函数 acos 计算 x 的三角反余弦函数主值。如果参数不在 [-1, +1] 范围内，则发生定义域错误。

返回值

函数 acos 返回 [0，π] 弧度范围内的反余弦值。

asin

7.5.2.2 函数 asin

概述

```
#include <math. h>
double asin(double x);
```

说明

函数 asin 计算 x 的三角反正弦函数主值。如果参数不在 [-1, +1] 范围内，则发生定义域错误。

返回值

函数 asin 返回 [− π/2, ＋ π/2] 弧度范围内的反正弦值。

atan

7.5.2.3 函数 atan

概述

```
#include <math.h>
double atan(double x);
```

说明

函数 atan 计算 x 的三角反正切函数主值。

返回值

函数 atan 返回 [$-\pi/2$, $+\pi/2$] 弧度范围内的反正切值。

atan2

7.5.2.4 函数 atan2

概述

```
#include <math.h>
double atan2(double y, double x);
```

说明

函数 atan2 计算 y/x 的三角反正切函数主值，根据两个参数的符号确定返回值的象限信息。若两个参数都为零，则发生定义域错误。

返回值

函数 atan2 返回 y/x 的反正切，在 [$-\pi$, $+\pi$] 弧度范围内。

cos

7.5.2.5 函数 cos

概述

```
#include <math.h>
double cos(double x);
```

说明

函数 cos 计算 x（以弧度为单位）的余弦值。

返回值

函数 cos 返回余弦值。

sin

7.5.2.6 函数 sin

概述

```
#include <math.h>
double sin(double x);
```

说明

函数 sin 计算 x（以弧度为单位）的正弦值。

返回值

函数 sin 返回正弦值。

tan

7.5.2.7 函数 tan

概述

```
#include <math.h>
double tan(double x);
```

说明

函数 tan 返回 x（以弧度为单位）的正切值。

返回值

函数 tan 返回正切值。

7.5.3 双曲函数

cosh

7.5.3.1 函数 cosh

概述

```
#include <math.h>
double cosh(double x);
```

说明

函数 cosh 计算 x 的双曲余弦。若 x 的数值太大，则发生值域错误。

返回值

函数 cosh 返回双曲余弦值。

sinh

7.5.3.2 函数 sinh

概述

```
#include <math.h>
double sinh (double x);
```

说明

函数 sinh 计算 x 的双曲正弦。如果 x 的数值太大，则发生值域错误。

返回值

函数 sinh 返回双曲正弦值。

tanh

7.5.3.3 函数 tanh

概述

```
#include <math.h>
double tanh(double x);
```

说明

函数 tanh 计算 x 的双曲正切。

返回值

函数 tanh 返回双曲正切值。

7.5.4 指数函数和对数函数

exp

7.5.4.1 函数 exp

概述

```
#include <math.h>
double exp(double x);
```

说明

函数 exp 计算 x 的指数函数。若 x 的数值太大，则发生值域错误。

返回值

函数 exp 返回指数值。

frexp

7.5.4.2 函数 frexp

概述

```
#include <math.h>
double frexp(double value, int *exp);
```

说明

函数 frexp 把一个浮点数分为一个规格化小数和一个 2 的整数幂。它把整数值存储在 exp 指向的 int 类型的对象中。

返回值

函数 frexp 返回值 x，使 x 为 [1/2, 1) 范围内的 double 值，或者为零，且 value 等于 x 乘以 2 的 *exp 次幂。如果 value 为零，则结果的两部分都为零。

ldexp

7.5.4.3 函数 ldexp

概述

```
#include <math.h>
double ldexp(double x, int exp);
```

说明

函数 ldexp 计算一个浮点数和 2 的整数幂的乘积，有可能发生值域错误。

返回值

函数 ldexp 返回 x 乘以 2 的 exp 次幂的值。

log

7.5.4.4 函数 log

概述

```
#include <math.h>
double log(double x);
```

说明

函数 log 计算 x 的自然对数。如果参数为负，则发生定义域错误；如果参数为零，则发生值域错误。

返回值

函数 log 返回自然对数值。

log10

7.5.4.5 函数 log10

概述

```
#include <math.h>
double log10 (double x);
```

说明

函数 log10 计算 x 的以 10 为底的对数。如果参数为负，则发生定义域错误；如果参数为零，则可能发生值域错误。

返回值

函数 log10 返回以 10 为底的对数。

modf

7.5.4.6 函数 modf

概述

```
#include <math.h>
double modf(double value, double *iptr);
```

说明

函数 modf 把参数 value 分为整数部分和小数部分，它们的符号和参数的相同。它把整数部分存储在 iptr 指向的 double 类型的对象中。

返回值

函数 modf 返回 value 的带符号的小数部分。

7.5.5 幂函数

pow

7.5.5.1 函数 pow

概述

```
#include <math.h>
double pow(double x, double y);
```

说明

函数 pow 计算 x 的 y 次幂。如果 x 为负值且 y 不是一个整数值，则发生定义域错误。当 x 为零且 y 小于等于零时，如果结果不能表示，则发生定义域错误。也可能发生值域错误。

返回值

函数 pow 返回 x 的 y 次幂。

sqrt

7.5.5.2 函数 sqrt

概述

```
#include <math.h>
double sqrt(double x);
```

说明

函数 sqrt 计算 x 的非负的平方根。如果参数为负，则发生定义域错误。

返回值

函数 sqrt 返回平方根的值。

7.5.6 取整函数、绝对值和取余函数

ceil

7.5.6.1 函数 **ceil**

概述

```
#include <math.h>
double ceil(double x);
```

说明

函数 ceil 计算不小于 x 的最小整数。

返回值

函数 ceil 返回不小于 x 的最小整数，表示为 double 类型。

fabs

7.5.6.2 函数 **fabs**

概述

```
#include <math.h>
double fabs(double x);
```

说明

函数 fabs 计算浮点数 x 的绝对值。

返回值

函数 fabs 返回 x 的绝对值。

floor

7.5.6.3 函数 **floor**

概述

```
#include <math.h>
double floor(double x);
```

说明

函数 floor 计算不大于 x 的最大整数。

返回值

函数 floor 返回不大于 x 的最大整数，表示为 double 类型。

fmod

7.5.6.4 函数 **fmod**

概述

```
#include <math.h>
double fmod(double x, double y);
```

说明

函数 fmod 计算 x/y 的浮点余数。

返回值

对某个整数值 *i*，函数 fmod 返回值 x-*i**y。因此，如果 y 不为零，结果的符号和 x 相同且数值上比 y 小；如果 y 为零，则或者发生定义域错误，或者函数 fmod 返回零，这是由实现定义的。

脚注

103. 参考“库的展望”（7.13.4）。

104. HUGE_VAL 在支持无穷大的实现上可以是正无穷大。

105. 在一个支持无穷大的实现下，如果函数的数学定义域不包括无穷大，就允许把无穷大作为一个定义域错误的参数对待。

7.3 <math.h> 的使用

在使用 <math.h> 中声明的函数之前，你应该对它们很熟悉因为很少有人会突然想计算一个角的余弦，所以对下面每个函数的常规解释都很少。

HUGE_VAL HUGE_VAL——这个宏通常会展开为一个非常大的 *double* 常量，它一般情况下和 <float.h> 中定义的 DBL_MAX 展开的值相等。在那些对无穷大（Inf）不提供特殊编码的机器上，通常认为返回这样的一个大值是警告发生了值域错误的最好方式。然而，即使被警告了，HUGE_VAL 也能和 Inf 相等。把一个数学函数的返回值和 HUGE_VAL 或者 -HUGE_VAL 相比较可能会很安全。（检测 errno 是否设置为 ERANGE 可能会更好，这两个宏都在 <errno.h> 中定义。）不要通过任何其他的方式使用 HUGE_VAL。

acos acos——函数 acos 和 asin 经常使用同一个函数来计算。每一个函数都在给出一个直角三角形的一条直角边和斜边的前提下计算出它的一个锐角值。因此，要注意 acos 的参数是一个比例量，特别是当它的一个参数具有 sqrt(1.0 - x * x) 这样的形式时更是如此。因此，最好去调用 asin、atan 或者 atan2。

asin asin——参考上面的 acos。

atan atan——函数 atan 和 atan2 经常使用一个公共函数来计算。然而，后者更常用。要优先使用后者，特别是当参数值是一个比值的时候。同样参考上面的 acos。

atan2 atan2——给定 *X-Y* 平面上的一个点坐标，该函数能有效地计算一个半径向量与 *X* 轴构成的角。目前它是 acos、asin、atan 和 atan2 这四个函数中最常用的函数。因此，要优先使用这个函数。

ceil ceil——函数 ceil、floor 和 modf 让你可以通过各种方式对一个浮点数的小数部分进行操作。使用这些函数要比把浮点数转换为整数类型安全得多，因为它们可以对任意的浮点值进行操作而不会造成溢出。注意，ceil 向 *X* 轴的右边取值，而 floor 向左边取值。想要得到和浮点值 x 最接近的整数，可以写：

```
x < 0.0 ? ceil(x - 0.5) : floor(x + 0.5)
```

cos cos——函数 cos 和 sin 经常使用同一个函数来计算。每一个函数都把它们的参数以 *X* 轴或者 *Y* 轴为中心减小到 π 弧度的范围。因此要注意 cos 的那些包含了 π/2 的整数倍的参数，这时可能会调用 sin 来代替它。可以省略参数中的任何 2*π 的倍数。函数对于消除 2*π 的倍数的工作可能会比手工做得好，但是，参数每增加一个 2*π，cos 的结果的精度几乎就会减少 3 位。

对于足够大的参数来说，即使这个函数没有报错，它的结果可能也没有什么意义了。

cosh　cosh——可以使用这个函数来代替下面的恒等式：

```
cosh(x) ≡ 0.5 * (exp(x) + exp(-x))
```

或者它的任何优化形式。和这个表达式不同，cosh会产生一个更精确的结果，并且它覆盖了函数值可表示的全部x的范围。

exp　exp——如果exp的参数具有y*log(x)的形式，就用pow(x,y)代替这个表达式。后者会更精确。

fabs　fabs——这个函数应该相当快。如果实现支持Inf和–Inf这样的特殊编码的话，对这样的参数，这个函数也会计算出正确的结果。

floor　floor——参考上面的ceil。

fmod　fmod——这个函数确定浮点运算中类似于整数除法中余数的值。有时候可以利用它通过重复迭代把一个参数缩减到一个子区域，因为fmod比直接减去该区间的一个倍数更好而且更安全。然而，本章后面描述的其他的技术在参数的缩减方面会做得更好。

frexp　frexp——当需要对一个浮点值的小数部分和指数部分独立地进行操作时，可以使用这个函数来把一个浮点值分为两部分。和它相对的函数是下面的ldexp。

ldexp　ldexp——在对一个浮点值的整数和指数部分分别进行操作之后，可以使用这个函数把它们重新组合起来。和它相对的函数是上面的frexp。

log　log——log(x)是自然对数，经常记做$\log_e(x)$或者$\ln(x)$。当然，可以通过乘以一个转换因子$\log_b(e)$(或者$1/\log_e(b)$得到x对任何底数b的对数值。

log10　log10——log10(x)通常由log(x)计算。如果要对log10的结果乘以一个转换因子，那么不妨考虑一下使用log来代替它。

modf　modf——当需要对一个浮点值的整数部分和小数部分分别进行操作时，可以使用这个函数对浮点值进行拆分。

pow　pow——这个函数通常是<math.h>中声明的所有函数中说明最详细的一个。一个好的实现会为pow(x,y)生成一个比它的等价形式exp(y*log(x))更好的结果，然而，它花费的时间可能也更长。当e是自然对数的底数时，可以用exp(y)代替pow(e,y)，用sqrt(x)代替pow(x,0.5)，用x*x代替pow(x,2.0)。

sin　sin——参考上面的cos。

sinh　sinh——使用这个函数来代替下面的恒等式：

```
sinh(x) ≡ 0.5 * (exp(x) - exp(-x))
```

或者它的任何优化形式。和这个表达式不同，sinh 应该产生一个更精确的结果，特别是对比较小的参数。这个函数也覆盖了函数值可以表示的 x 的所有范围。

sqrt sqrt——这个函数通常比它的等价式 pow(x,0.5) 快得多。

tan tan——这个函数首先把它的参数缩小到以 *X* 轴为中心的 π 弧度的范围内，因此可以省略任何加到参数上的 2*π 的倍数。函数在消除 2*π 的倍数方面可能要比手工做得好。但是，参数每增加一个 2*π，tan 的结果的精度就几乎减少 3 位。对于很大的参数来说，即使这个函数没有报错，它的结果可能也没有什么意义了。

tanh tanh——使用这个函数代替恒等式：

```
tanh(x) ≡ (exp(2.0 * x) - 1.0) / (exp(2.0 * x) + 1.0)
```

或者它的任何优化形式。和这个表达不同，这个函数应该产生一个更精确的结果，特别是对比较小的参数。这个函数也覆盖了函数值可以表示的所有的 x 的范围。

7.4 <math.h> 的实现

<math.h> 中包含了各种各样的函数。我把它们分为 3 组进行讨论。

- 对一个浮点值的组成部分进行操作的半数字函数，这些组成部分包括指数部分、整数部分和小数部分等。
- 三角函数和反三角函数。
- 指数函数、对数函数和特殊的幂函数。

原语 在这个过程中我也给出了几个低级原语，<math.h> 中声明的所有函数都使用这些函数，这样，它们就不会依赖于浮点值的特殊表示。7.1 节讨论了由这些特殊的原语集合决定的机器的一般属性。我要再强调一次，参数化并没有覆盖现代计算机中使用的所有的浮点表示。所以有时候必须为某些计算机体系结构修改一个或者更多的原语，甚至可能会修改更高级别的函数。

头文件 <math.h> 图 7-1 显示了文件 math.h，这个文件中有些让人不太明白的地方。其中一点是隐式宏。因为文件中的某些数学函数依次调用其他的函数，而这些隐式宏消除了一次函数调用。

宏 HUGE_VAL 还有一点是宏 HUGE_VAL。它被定义为 IEEE 754 的 +Inf 编码。为了实现这一点，我引入了类型 _Dconst，它是一个联合，且允许把一个数据对象初始化为由四个 *unsigned short* 类型的值组成的数组，然后把这个数据对象作为 *double* 类型访问（参考 4.4 节，一个相似的技巧）。数据对象 _Hugeval 是以这种方式构建起来的几个浮点值之一。

图 7-1
math.h

```
/* math.h standard header */
#ifndef _MATH
#define _MATH
        /* macros */
#define HUGE_VAL _Hugeval._D
        /* type definitions */
typedef const union {
    unsigned short _W[4];
    double _D;
    } _Dconst;
        /* declarations */
double acos (double);
double asin (double);
double atan (double);
double atan2 (double,double);
double ceil (double);
double cos (double);
double cosh (double);
double exp (double);
double fabs (double);
double floor (double);
double fmod (double,double);
double frexp (double,int *);
double ldexp (double,int);
double log (double);
double log10 (double);
double modf (double,double*);
double pow (double,double);
double sin (double);
double sinh (double);
double sqrt (double);
double tan (double);
double tanh(double);
double _Asin (double,int);
double _Log (double,int);
double _Sin (double,unsigned int);
extern _Dconst _Hugeval;
        /* macro overrides */
#define acos (x) _Asin (x, 1)
#define asin (x) _Asin (x, 0)
#define cos (x) _Sin (x, 1)
#define log(x) _Log(x, 0)
#define log10 (x) _Log(x, 1)
#define sin (x) _Sin(x, 0)
#endif
```
□

图 7-2
xvalues.c

```
/* values used by math functions -- IEEE 754 version */
#include "xmath. h"

        /* macros */
#define NBITS (48+_DOFF)
#if _D0
#define INIT (w0) 0, 0, 0, w0
#else
#define INIT(w0) w0, 0, 0, 0
#endif
        /* static data */
_Dconst_Hugeval = {{INIT(-DMAX<<_DOFF)}};
_Dconst_Inf = {{INIT(_DMAX<<_DOFF)}};
_Dconst_Nan = {{INIT(_DNAN)}};
_Dconst_Rteps = {{INIT((_DBIAS-NBITS/2)<<_DOFF)}};
_Dconst_Xbig = {{INIT((_DBIAS+NBITS/2) <<_DOFF)}};
```
□

_Hugeval_Inf

图 7-2 显示了文件 xvalues.c，它定义了一些值。该文件包含了和 _Hugeval 相对应的 _Inf 的定义。我提供这两个值，以供修改 HUGE_VAL 的定义。这个文件也定义了如下几个值：

_Nan
- _Nan，当操作数都不是 NaN 时，函数返回的 NaN 的编码。

_Rteps
- _Rteps，DBL_EPSILON 的平方根（近似值），某些函数使用它在不同的近似值之间作出选择。

_Xbig
- _Xbig，_Rteps._D 的相反数，某些函数使用它在不同的近似值之间作出选择。

当你知道函数是怎样使用最后两个值的时候，这两个值的作用也就体现出来了。

头文件 <yvals.h>

实际上文件 xvalues.c 的可读性很差，它参数化的形式和图 4-3 的文件 xfloat.c 很相似。这两个文件都利用了内部头文件 <yvals.h> 中定义的依赖于具体系统的参数。

头文件 "xmath.h"

xvalues.c 并没有直接地包含 <yvals.h>，而是包含了内部头文件 "xmath.h"，而该头文件又包含了 <yvals.h>。所有实现 <math.h> 的文件都包含了 "xmath.h"。因为该文件包含的内容太零散，所以我只在必要的时候才给出相关部分内容。在图 7-38 中有 "xmath.h" 的一个完整的列表。下面是 "xmath.h" 中定义的和 xvalues.c 相关的宏：

```
#define _DFRAC ((1<<_DOFF)-1)
#define _DMASK (0 x7fff&~_DFRAC)
#define _DMAX((1<<(15-_DOFF))-1)
#define _DNAN (0x8000|_DMAX<<_DOFF|1<<(_DOFF-1))
```

整理上面这些看起来没有意义的东西，就会发现：

- Inf 的编码具有所有小数位为零的最大可能的特征值（_DMAX）。
- 生成的 NaN 的编码拥有小数位的最高有效位已经被设置的最大可能的特征值。

图 7-3
fabs.c

```
/* fabs function */
#include "xmath.h"

double (fabs)(double x)
    {                                           /* compute fabs */
    switch (_Dtest (&x) )
        {                               /* test for special codes */
    case NAN:
        errno = EDOM;
        return (x) ;
    case INF:
        errno = ERANGE;
        return (_Inf._D) ;
    case 0:
        return (0.0) ;
    default:                                          /* finite */
        return (x < 0.0 ? -x : x) ;
        }
    }
```

一般来说，NaN 应该至少有一个非零的小数位。对 NaN 使用这种特殊的编码是为了和 Intel 80X87 数学协处理器相匹配。

函数 fabs　所有这些编码的出现使最简单的数学函数变得不简单了。例如，图 7-3 显示了文件 fabs.c，在比较简单的情况下，可以把它概括为最后一个返回语句：

```
return (x < 0.0 ? -x : x);
```

然而，在这里，在处理参数 x 的有限值和零同时，我们还想要恰当地处理 NaN、–Inf 和 +Inf。这就会花费更多的判断语句。

函数 _Dtest　图 7-4 显示了文件 xdtest.c，它定义了对一个 *double* 值进行分类的函数 _Dtest。内部头文件 "xmath.h" 定义了 _Dtest 使用的各种偏移量和类别值，这里相关的宏定义有：

```
        /* word offsets within double */
#if _D0==3
#define _D1      2      /* little-endian order */
#define _D2      1
#define _D3      0
#else
#define _D1      1      /* big-endian order */
#define _D2      2
#define _D3      3
#endif
         /* return values for _D functions */
#define FINITE   -1
#define INF      1
#define NAN      2
```

注意一个特征值为零的浮点值不一定为零，IEEE 754 支持渐进的下溢，一个值只有在所有位（除了符号位）都是零的时候才是零。

图 7-4 xdtest.c

```
/* _Dtest function -- IEEE 754 version */
#include "xmath.h"

short _Dtest (double *px)
    {                                           /* categorize *px */
    unsigned short *ps = (unsigned short *)px;
    short xchar = (ps[_D0] & _DMASK) >> _DOFF;

    if (xchar ==_DMAX)                               /* NaN or INF */
        return (ps[_D0] & _DFRAC || ps[_D1]
            || ps[_D2] || ps[_D3] ? NAN : INF);
    else if (0 < xchar || ps[_D0] & _DFRAC
        || ps[_D1] || ps[_D2] || ps [_D3])
        return (FINITE);                                /* finite */
    else
        return (0);                                       /* zero */
    }
```

ceil floor 图 7-5 显示了文件 ceil.c，图 7-6 显示了文件 floor.c。这些文件中定义的每一个函数都要把它的参数 x 的所有小数部分都设置为零，而且都需要知道小数部分的初始值是不是为非零。然后每一个函数通过略微不同的方式对剩余的整数部分进行调整。

函数 _Dint 图 7-7 显示了文件 xdint.c，该文件定义了函数 _Dint。如果 *px 是一个有限值，该函数就对所有小于阈值的小数位进行测试和清零。那个阈值是 2 的 xexp 次幂。(其他的函数也有可能用 xexp 的值而不是零来调用 _Dint。) 对小数位进行清零的代码也有一定的技巧。

注意 ps[sub[xchar]] 中在一个下标中使用另一个下标的用法。下标 sub[xchar] 纠正了不同的计算机体系结构下浮点值设计的差别。*switch* 语句包括了一连串的 *case* 标号，这通常是一种容易被误解、不太明智的做法。然而，为了提高性能，这两种方法我都用了。

图 7-5 ceil.c

```
/* ceil function */
#include "xmath.h"

double (ceil) (double x)
    {                                           /* compute ceil(x) */
    return (_Dint(&x, 0) < 0 && 0.0 < x ? x + 1.0 : x) ;
    }
```

图 7-6 floor.c

```
/* floor function */
#include "xmath.h"

double (floor) (double x)
    {                                          /* compute floor(x) */
    return (_Dint(&x, 0) < 0 && x < 0.0 ? x - 1.0 : x);
    }
```

图 7-7
xdint.c

```
/* _Dint function -- IEEE 754 version */
#include "xmath.h"

short _Dint(double *px, short xexp)
    {                    /* test and drop (scaled) fraction bits */
    unsigned short *ps = (unsigned short * )px;
    unsigned short frac = ps[_D0] & _DFRAC
        || ps[_D1] || ps[_D2] || ps[_D3];
    short xchar = (ps[_D0] & _DMASK) >> _DOFF;

    if (xchar == 0 && !frac)
        return (0);                                    /* zero */
    else if (xchar != _DMAX)
        ;                                              /* finite */
    else if (!frac)
        return (INF) ;
    else
        {                                              /* NaN */
        errno = EDOM;
        return (NAN) ;
        }
    xchar = (_DBIAS+48+_DOFF+1) - xchar - xexp;
    if (xchar <= 0)
        return (0);                       /* no frac bits to drop */
    else if ((48+_DOFF) < xchar)
        {                                        /* all frac bits */
        ps[_D0] = 0, ps[_D1] = 0;
        ps[_D2] = 0, ps[_D3] = 0;
        return (FINITE);
        }
    else
        {                                 /* strip out frac bits */
        static const unsigned short mask[] = {
            0x0000, 0x0001, 0x0003, 0x0007,
            0x000f, 0x001f, 0x003f, 0x007f,
            0x00ff, 0x01ff, 0x03ff, 0x07ff,
            0x0fff, 0x1fff, 0x3fff, 0x7fff};
        static const size_t sub[] = {_D3, _D2, _D1, _D0};

        frac = mask[xchar & 0xf];
        xchar >>= 4;
        frac &= ps[sub[xchar]];
        ps[sub[xchar]] ^= frac;
        switch (xchar)
            {                               /* cascade through! */
        case 3:
           frac |= ps[_D1], ps[_D1] = 0;
        case 2 :
           frac |= ps[_D2], ps[_D2] = 0;
        case 1 :
           frac |= ps[_D3], ps[_D3] = 0;
           }
        return (frac ? FINITE : 0);
        }
    }
```
□

图 7-8 modf.c

```
/* modf function */
#include "xmath.h"

double (modf) (double x, double *pint)
    {                                   /* compute modf(x, &intpart) */
    *pint = x;
    switch (_Dint(pint, 0))
        {                                   /* test for special codes */
    case NAN:
        return (x);
    case INF:
    case 0:
        return (0.0);
    default :                                           /* finite */
        return (x - *pint);
        }
    }
```

函数 modf

图 7-8 显示了文件 modf.c。它定义了函数 modf，该函数只是比 ceil 和 floor 看起来复杂一点。和那些函数一样，modf 也依靠函数 _Dint 来完成那些困难的部分。

函数 frexp

图 7-9 显示了文件 frexp.c。它定义了函数 frexp，这个函数从一个有限大小的参数 x 中提取出它的指数。又一次，一个相当简单的函数由于各种特殊编码的存在而变得复杂了。又一次，一个更加灵活的低级函数完成了大部分困难的工作。

图 7-9 frexp.c

```
/* frexp function */
#include "xmath.h"

double (frexp)(double x, int *pexp)
    {                                       /* compute frexp(x, &i) */
    short binexp;

    switch (_Dunscale (&binexp, &x))
        {                                   /* test for special codes */
    case NAN:
    case INF:
        errno = EDOM;
        *pexp = 0;
        return (x) ;
    case 0:
        *pexp = 0;
        return (0.0);
    default:                                            /* finite */
        *pexp = binexp;
        return (x);
        }
    }
```

图 7-10 ldexp.c

```
/* ldexp function */
#include "xmath.h"

double (ldexp)(double x, int xexp)
    {                                       /* compute ldexp(x, xexp) */
    switch (_Dtest(&x))
        {                                   /* test for special codes */
    case NAN:
        errno = EDOM;
        break;
    case INF:
        errno = ERANGE;
        break;
    case 0:
        break;
    default:                                                /* finite */
        if (0 <= _Dscale(&x, xexp))
            errno = ERANGE;
        }
    return (x);
    }
```

图 7-11 xdunscal.c

```
/* _Dunscale function -- IEEE 754 version */
#include "xmath.h"

short _Dunscale(short *pex, double *px)
    {           /* separate *px to 1/2 <= |frac| < 1 and 2^*pex */
    unsigned short *ps = (unsigned short *)px;
    short xchar = (ps[_D0] & _DMASK) >> _DOFF;

    if(xchar == _DMAX)
       {                                            /* NaN or INF */
       *pex = 0;
       return(ps[_D0]& _DFRAC || ps[_D1]
       || ps[_D2] || ps[_D3] ? NAN : INF);
       }
    else if (0 < xchar || (xchar = _Dnorm(ps))!= 0)
       {                               /* finite,reduce to [1/2,1) */
       ps[_D0] = ps[_D0] & ~_DMASK | _DBIAS << _DOFF;
       *pex = xchar - _DBIAS;
       return (FINITE);
       }
    else
       {                                                /*zero */
       *pex = 0;
       return(0);
       }
    }
```

图 7-12 xdscale.c

```
/* _Dscale function -- IEEE 754 version */
#include "xmath.h"

short _Dscale(double *px, short xexp)
    {                           /* scale *px by 2^xexp with checking */
    long lexp;
    unsigned short *ps = (unsigned short *)px;
    short xchar = (ps[_D0] 6 _DMASK) >> _DOFF;

    if (xchar == DMAX)                                  /* NaN or INF */
        return (ps[_D0] 6 _DFRAC || ps[_D1]
            || ps[_D2] || ps[_D3] ? NAN : INF);
    else if (0 < xchar)
                                                            /* finite */
    else if ((xchar = _Dnorm(ps)) == 0)
       return (0);                                            /* zero */
    lexp = (long)xexp + xchar;
    if (_DMAX <= lexp)
        {                                /* overflow , return +/-INF */
        *px = ps[_D0] 6 _DSIGN ? -_Inf._D : _Inf._D;
        return (INF);
        }
    else if (0 < lexp)
        {                                  /* finite result, repack */
        ps [_D0] = ps [_D0] 6 ~_DMASK | (short)lexp << _DOFF;
        return (FINITE) ;
        }
    else
        {                                  /* denormalized, scale */
        unsigned short sign = ps [_D0] 6 _DSIGN;

        ps[_D0] = 1 << _DOFF | ps[_D0] 6 _DFRAC;
        if (lexp < -(48+_DOFF+1))
            xexp = -1;                           /* certain underflow */
        else
           {                                  /* might not underflow */
           for (xexp = lexp: xexp <= -16; xexp += 16)
               {                                    /* scale by words */
               ps [_D3] = ps[_D2], ps[_D2] = ps[_D1];
               ps[_D1] = ps[_D0],ps[_D0] = 0;
               }
           if ((xexp = -xexp) != 0)
               {                                     /* scale by bits */
               ps[_D3] = ps[_D3] >> xexp
                   | ps[_D2] << 16 - xexp;
               ps[_D2] = ps[_D2] >> xexp
                   | ps[_D1] << 16 - xexp;
               ps[_D1] = ps[_D1] >> xexp
                   | ps[_D0] << 16 - xexp;
               ps[_D0] >>= xexp;
               }
            }
```

图 7-12
（续）

```
  if (0 <= xexp && (ps[_D0] || ps[_D1]
      || ps[_D2] || ps[_D3]))
      {                                       /* denormalized */
      ps[_D0] | = sign;
      return (FINITE);
      }
  else
      {                                /* underflow, return +/-0 */
      ps[_D0] = sign, ps[_D1] = 0;
      ps[_D2] = 0, ps[_D3] = 0 ;
      return (0);
      }
   }
}
```

图 7-13
xdnorm.c

```
/* _Dnorm function -- IEEE 754 version */
#include "xmath.h"

short _Dnorm(unsigned short *ps)
    {                             /* normalize double fraction */
    short xchar;
    unsigned short sign = ps [_D0] & _DSIGN;

    xchar = 0;
    if ((ps[_D0] &= _DFRAC) != 0 || ps[_D1]
        || ps[_D2] || ps[_D3])
        {                                     /* nonzero, scale */
        for (; ps [_D0] == 0; xchar -= 16)
            {                               /* shift left by 16 */
            ps [_D0] = ps[_D1], ps[_D1] = ps[_D2];
            ps [_D2] = ps[_D3], ps[_D3] = 0;
            }
        for (; ps[_D0] < 1<<_DOFF; --xchar)
            {                                /* shift left by 1 */
            ps[_D0] = ps[_D0] << 1 | ps[_D1] >> 15;
            ps[_D1] = ps[_D1] << 1 | ps[_D2] >> 15;
            ps[_D2] = ps[_D2] << 1 | ps[_D3] >> 15:
            ps[_D3] <<= 1;
            }
        for (; 1<<_DOFF+1 <= ps[_D0]; ++xchar)
            {                               /* shift right by 1 */
            ps[_D3] = ps[_D3] >> 1 | ps[_D2] << 15;
            ps[_D2] = ps[_D2] >> 1 | ps[_D1] << 15;
            ps[_D1] = ps[_D1] >> 1 | ps[_D0] << 15;
            ps[_D0] >>= 1;
            }
        ps[_D0] &= _DFRAC;
        }
    ps[_D0] |= sign;
    return (xchar);
    }
```

图 7-14
fmod.c

```
/* fmod function */
#include "xmath.h"

double (fmod)(double x, double y)
    {                                   /* compute fmod(x, y) */
    const short errx = _Dtest(&x);
    const short erry = _Dtest(&y);

    if (errx == NAN || erry == NAN || errx == INF || erry == 0)
        {                                   /* fmod undefined */
        errno = EDOM;
        return (errx == NAN ? x : erry == NAN ? y : _Nan._D);
        }
    else if (errx == 0 || erry == INF)
        return (x);         /* fmod(0, nonzero) or fmod(finite,INF) */
    else
        {                                   /* fmod(finite, finite) */
        double t;
        short n, neg, ychar;

        if (y < 0.0)
            y = -y;
        if (x < 0.0)
            x = -x, neg = 1;
        else
            neg = 0;
        for (t = y, _Dunscale(&ychar, &t), n = 0; ;)
            {                           /* subtract |y| until |x|<|y| */
            short xchar;

            t = x;
            if (n < 0 || _Dunscale(&xchar, &t ) == 0
                || (n = xchar - ychar) < 0)
                return (neg ? -x : x) ;
            for (; 0 <= n; --n)
                {                       /* try to subtract |y|*2^n */
                t = y, _Dscale(&t, n);
                if (t <= x)
                    {
                    x -= t;
                    break;
                    }
                }
            }
        }
    }
```
□

函数 ldexp

图 7-10 显示了文件 ldexp.c。函数 ldexp 面临着和 frexp 相似的问题，只不过它们的功能相反。该函数一旦发送了任何特殊的编码，就还要执行一个重要的任务。函数也要调用一个低级函数。这两个低级别的函数如下所示。

函数 _Dunscale

图 7-11 显示了文件 xdunscal.c。它定义了函数 _Dunscale，这个函数把 _Dtest 和 frexp 的行为组合到一起，这样，其他几个数学函数就可以更加方便地使用它们了。通过调用 _Dunscale，函数 frexp 要完成的工作就很少了。

在不出现渐进下溢的情况下，_Dunscale 本身只需要完成一个相当简单的工作。一个标准化的值有一个非零的特征值，而且在给出的小数的最高有效位的左边还有一个隐藏的小数位。一个零特征值和一个前面没有隐藏位的非零小数标志着渐进下溢。这两种形式都要转换为区间 [0.5,1.0) 中的一个标准化的小数和一个合适的二元指数。下面讲到的函数 _Dnorm 会处理这种繁琐的工作。

函数 _Dscale

图 7-12 显示了文件 xdscale.c，这个文件定义了函数 _Dscale。因为它可以通过其他方式被调用，所以它也会受到特殊编码的困扰。把一个很小的值 xexp 加到一个有限的 *px 的指数上可能会导致上溢、渐进下溢或者下溢。在形成新的指数的时候，甚至要考虑到整数溢出。这就是函数首先以 *long* 类型计算它们的和的原因。

_Dscale 函数的复杂性主要在于产生一个渐进下溢。这个操作实际上是 _Dnorm 的反操作。

函数 _Dnorm

图 7-13 显示了文件 xdnorm.c，这个文件定义了函数 _Dnorm。它把一个渐进下溢的小数部分进行标准化并且相应地调整特征值。为了提高性能，函数在任何可能的时候会把小数部分一次左移 16 位。这就是它要准备好每次不仅左移一位也要右移一位的原因。有时候它可能会移过头而不得不倒回来。

函数 fmod

图 7-14 显示了文件 fmod.c。函数 fmod 是 <math.h> 中声明的半数值函数中的最后一个，也是最复杂的一个。原则上，它重复地从 x 的值中减去 y 的值，直到剩下的数比 y 小。实际上，这样做花费的时间很长，即使它可以保留任何精度，这也是不能容忍的。

因此 fmod 实际上做的是在每一次减之前都把 y 乘以 2 的最大可能的幂。这仍然要求很多次的迭代，但结果会相当精确。注意 fmod 使用 _Dscale 和 _Dunscale 操作指数的方式。它使用 _Dunscale 来提取 x 和 y 的指数以对它们的数值执行一个快速但是很粗糙的比较。如果 fmod 确定一个减法能行得通，它就使用 _Dscale 来把 x 扩大到近似合适的大小。

函数 _Sin 现在让我们看看三角函数。图 7-15 显示了文件 xsin.c，该文件定义了函数 _Sin。如果 qoff 为 0 函数就计算 sin(x)；如果 qoff 为 1，函数就计算 cos(x)。计算余弦时使用这样的“象限偏移”避免了把 π/2 加到参数上时造成的精度丢失。在用多项式对三角函数逼近时，我使用了截断的泰勒展式，并且为了改善逼近的误差分布，还使用了切比雪夫多项式对泰勒展式进行“节约”。(如果不明白这是什么意思也没关系。)

把参数减小到区间 [−π/4, π/4] 的时候一定要谨慎。确定应该从参数中减去多少个 π/2 非常简单，这样就确定了 quad，即该角所在的象限（位于 4 个轴其中的一个的中间）。需要用 quad + qoff 的较低的两位来确定是计算余弦还是正弦以及是否需要对结果取负。注意将有符号的象限转换为无符号值的方式，这样负参数在所有的计算机体系结构下就能够得到一致的对待。

在这一点上要做的是以任意的精度计算 quad*π/2。要从参数中减去这个值并且在最高有效位忽略后仍然具有完整的 *double* 类型值。由于浮点值的取值范围很广，因此这是一个很高的目标，很难实现，而且也有点愚蠢。就像我在 7.3 节所说的那样，参数的数值越大，三角函数就变得越来越粗糙。除了某些值之外，其他所有的值都不能从 π/2 的整数倍中区分出来。有些人认为这是一种错误的情况，但是 C 标准不这样认为。循环函数对所有有限的参数值都要返回一些有意义的值，并且不能报告错误。

7

宏 HUGE_RAD 我选择了把这些区别分开来讲。通过在某些地方改编 Cody 和 Waite 使用的方法，我把 π/2 表示为“1.5”倍的 *double* 精度。头文件 "xmath.h" 把宏 HUGE_RAD 定义为：

```
#define HUGE_RAD    3.14e30
```

可以通过把一个参数除以 π/2 来使它达到这个值，然后就可以得到一个能转换为 *long* 类型的值，而且不用担心溢出。常数 c1 把 π/2 的最高有效位表示为 *double* 类型，当然这个 *double* 类型的低 32 位都是零。(常数 c2 提供了一个具有额外的精度的全 *double* 值)

这就意味着可以用一个任意的 *long*（转换为 *double*）乘以 c1 并且得到一个精确的结果。因此，只要参数的数值比 HUGE_RAD 小，就可以得到全 *duoble* 精度的缩减的参数。在下面的表达是中就是如此：

```
g = (x - g * c1) - g * c2;
```

对于比 HUGE_RAD 大的参数，这个函数可以简单地减去 2*π 的一个倍数。注意在分离 *double* 值的整数部分的过程中对 _Dint 的使用。或者可以说，一旦参数比这个值大了很多倍，那么函数 sin 和 cos 就不再强调精度。我认为对这种很大的参数考虑高精度就不值得了。

图 7-15
xsin.c

```
/* _Sin function */
#include "xmath .h"

/* coefficients */
static const double c[8] ={
    -0.000000000011470879,
     0.000000002087712071,
    -0.000000275573192202,
     0.000024801587292937,
    -0.001388888888888893,
     0.041666666666667325,
    -0.500000000000000000,
     1.0};
static const double s[8] = {
    -0.000000000000764723,
     0.000000000160592578,
    -0.000000025052108383,
     0.000002755731921890,
    -0.000198412698412699,
     0.008333333333333372,
    -0.166666666666666667,
     1.0};
static const double c1 = {3294198.0 / 2097152.0};
static const double c2 = {3.139164786504813217e-7};
static const double twobypi = {0.63661977236758134308};
static const double twopi = {6.28318530717958647693};

double _Sin (double x, unsigned int qoff)
    {                                       /* compute sin(x) or cos(x) */
    switch (_Dtest(&x))
        {
    case NAN:
        errno = EDOM;
        return (x);
    case 0:
        return (qoff ? 1.0 : 0.0);
    case INF:
        errno = EDOM;
        return (_Nan._D) ;
    default:                                            /* finite */
        {                                       /* compute sin/cos */
        double g;
        long quad;

        if (x < -HUGE_RAD || HUGE_RAD < x)
            {                           /* x huge, sauve qui peut */
            g = x / twopi;
            _Dint(&g, 0);
            x -= g * twopi;
            }
        g = x * twobypi;
        quad = (long) (0 < g ? g + 0.5 : g - 0.5);
        qoff += (unsigned long)quad & 0x3;
        g = (double)quad;
        g = (x - g * c1) - g * c2;
```

图 7-15
（续）

```
        if ((g < 0.0 ? -g : g) < _Rteps._D)
            {                   /* sin (tiny) ==tiny, cos (tiny)==1 */
            if (qoff & 0x1)
                g = 1.0;                                /* cos(tiny) */
            }
        else if (qoff & 0x1)
            g = _Poly(g * g, c,  7);
        else
            g *= _Poly(g * g, s,  7);
        return (qoff & 0x2 ? -g : g);
         }
        }
    }
```

图 7-16
xpoly.c

```
/* _Poly function */
#include "xmath.h"

double _Poly(double x, const double *tab, int n)
    {                                           /* compute polynomial */
    double y;

    for (y = *tab; 0 <= --n; )
        y = y * x + *++tab;
    return (y);
    }
```

_Sin 函数的其他部分就很简单了。如果缩减后的角 g 充分小，那么计算一个多项式的近似值就是在浪费时间；如果缩减的角确实很小，那么计算平方参数 g*g 的时候很可能产生下溢。这里，当 g*g 小于 <float.h> 中定义的 DBL_EPSILON 时，就称得上是“充分小”。注意，要加速这个测试，要使用 *double* 常量 _Rteps._D。

_Poly 图 7-16 显示了文件 xpoly.c，这个文件定义了函数 _Poly。函数 _Sin 使用 _Poly 并使用霍纳规则来计算一个多项式的值。

cos sin 图 7-17 显示了文件 cos.c，图 7-18 显示了文件 sin.c。这些文件定义了三角函数 cos 和 sin。头文件 <math.h> 为这两个函数都定义了屏蔽宏。

函数 tan 图 7-19 显示了文件 tan.c。函数 tan 和其他的三角函数 sin 和 cos 非常相似。它也把它的参数缩减到 [$-\pi/4$, $\pi/4$] 范围内。主要的区别是函数在这个缩减的区间上求近似值的方式。因为正切在 $\pi/2$ 的整数倍上有极值，所以它使用多项式的一个比例来求近似值会更好。Cody 和 Waite 提供了这些系数。

函数 _Asin 现在考虑一下三角函数的反函数。图 7-20 显示了文件 xasin.c，这个文件定义了函数 _Asin。如果 qoff 是零它就计算 asin(x)，如果 qoff 为 1 就计算 acos(x)。这就避免了对 acos 的结果修改两次。

图 7-17 cos.c

```
/* cos function */
#include <math.h>

double (cos)(double x)
    {                                          /* compute cos */
    return (_Sin(x, 1));
    }
```

图 7-18 sin.c

```
/* sin function */
#include <math. h>

double (sin)(double x)
    {                                          /* compute sin */
    return (_Sin(x, 0));
    }
```

_Asin 首先确定了参数的数值 y。它用 5 种不同的方式计算中间结果（也放在 y 中）。

- 如果 y<_Rteps._D，就使用参数本身。
- 否则，如果 y<0.5，就使用 Cody 和 Waite 给出的多项式近似法的一个比例。
- 否则，如果 y<1.0，使用同样的近似法来计算 2*asin(sqrt(1-x)/2))（有效地计算）。实际的算术会尽量减少中间值的有效值丢失。
- 否则，如果 y= =1.0，使用 0。
- 否则，y>1.0，函数报告一个定义域错误。

对这样零散的方法，人们所关心的是在边界处引入不连续性。在这种情况下，最让人担心的边界在 y 等于 0.5 的地方。

在这个过程中，_Asin 根据 idx 中的记号来确定最终的结果：

- 如果 idx&1，要求计算的是反余弦，不是反正弦；
- 如果 idx&2，参数是负的。
- 如果 idx&4，参数的数值大于 0.5。

最后的修正包括添加 π/4 的各个整数倍和取结果的负值。要进行阶段性求和，以防止有效值的丢失。

acos asin 图 7-21 显示了文件 acos.c，图 7-22 显示了文件 asin.c。这些文件定义了平凡函数 acos 和 asin。头文件 <math.h> 为这两个函数定义了屏蔽宏。

atan atan2 最后一个反三角函数是反正切，它有两种形式：atan(x) 和 atan2(y,x)。这两种形式都调用一个公共的函数 _Atan 来执行实际的计算。然而，和前面的三角函数不同，这个公共函数刚出现时并不是最好的选择。图 7-23 显示了文件 atan.c。图 7-24 显示了文件 atan2.c，它定义了函数 atan2，这个函数揭示了这 3 个函数协同工作的原理。

图 7-19 tan.c

```
/* tan function */
#include "xmath.h"

/* coefficients, after Cody & Waite, Chapter 9 */
static const double p[3] = {
    -0.17861707342254426711e-4,
     0.34248878235890589960e-2,
    -0.13338350006421960681e+0};
static const double q[4] = {
     0.49819433993786512270e-6,
    -0.31181531907010027307e-3,
     0.25663832289440112864e-1,
    -0.46671683339755294240e+0};
static const double c1 = {3294198.0 / 2097152.0};
static const double c2 = {3.139164786504813217e-7};
static const double twobypi = {0.63661977236758134308};
static const double twopi = {6.28318530717958647693};

double tan (double x)
    {                                           /* compute tan(x) */
    double g, gd;
    long quad;

    switch (_Dtest(&x))
        {
    case NAN:
        errno = EDOM;
        return (x);
    case INF:
        errno = EDOM;
        return (_Nan._D) ;
    case 0:
        return (0.0) ;
    default :                                           /* finite */
        if (X < -HUGE_RAD || HUGE_RAD < x)
            {                           /* x huge, sauve qui peut */
            g = x / twopi;
            _Dint(&g,0);
            x -= g * twopi;
            }
        g = x * twobypi;
        quad = (long)(0 < g ? g + 0.5 : g - 0.5);
        g = (double)quad;
        g = (x - g * c1) - g * c2;
        gd = 1.0;
        if (_Rteps._D < (g < 0.0 ? -g : g))
            {                              /* g*g worth computing */
            double y = g * g;

            gd+= (((q[0] * Y + q[1]) * Y +q[2]) * Y +q[3]) * Y;
            g += ((p[0] * Y + p[1]) * Y + p[2]) * Y * g;
            }
        return ((unsigned int)quad & 0x1 ? -gd / g : g / gd);
        }
    }
```
□

图 7-20
xasin.c

```
/* _Asin function */
#include "xmath.h"

/* coefficients, after Cody & Waite, Chapter 10 */
static const double p[5] = {
    -0.69674573447350646411e+0,
     0.10152522233806463645e+2,
    -0.39688862997504877339e+2,
     0.57208227877891731407e+2,
    -0.27368494524164255994e+2};
static const double q[6] = {
     0.10000000000000000000e+1,
    -0.23823859153670238830e+2,
     0.15095270841030604719e+3,
    -0.38186303361750149284e+3,
     0.41714430248260412556e+3,
    -0.16421096714498560795e+3};
static const double piby2 = {1.57079632679489661923} ;
static const double piby4 = {0.78539816339744830962};

double _Asin(double x, int idx)
    {                                   /* compute asin(x) or acos(x) */
    double g, y;
    const short errx = _Dtest(&x);

    if (0 < errx)
        {                                                /* INF, NaN */
        errno = EDOM;
        return (errx == NAN ? x : _Nan._D);
        }
    if (x < 0.0)
        y = -x, idx |= 2;
    else
        y = x;
    if (y < _Rteps._D)
        ;
    else if (y < 0.5)
        {                                   /* y*y worth computing */
        g = y * y;
        y += y * g * _Poly(g, p, 4) / _Poly(g, q, 5);
        }
    else if (y < 1.0)
        {                              /* find 2*asin(sqrt((1-x)/2)) */
        idx |= 4;
        g = (1.0 - y) / 2.0;                           /* NOT * 0.5! */
        y = sqrt(g);
        y += y;
        y += y * g * _Poly(g, p. 4) / _Poly(g, q, 5);
        }
    else if (y == 1.0)
        idx |= 4, y = 0.0;
    else
        {                                  /* 1.0 < |x|, undefined */
        errno = EDOM;
        return (_Nan._D);
```

图 7-20 (续)

```
        }
    switch (idx)
        {                                            /* flip and fold */
    default:                                        /* shouldn't happen */
    case 0:                                           /* asin, [0, 1/2) */
    case 5:                                           /* acos, (1/2, 1] */
        return (y);
    case 1:                                           /* acos. [0, 1/2) */
    case 4:                                           /* asin, (1/2, 1] */
        return ((piby4 - y) + piby4);
    case 2:                                         /* asin, [-1/2, 0) */
        return (-y) ;
    case 3:                                         /* acos, [-1/2, 0) */
        return ((piby4 + y) + piby4);
    case 6:                                        /* asin, [-1, -1/2) */
        return ((-piby4 + y) - piby4);
    case 7:                                        /* acos, [-1, -1/2) */
        return ((piby2 - y) + piby2);
        }
    }
```

图 7-21 acos.c

```
/* acos function */
#include <math.h>

double (acos) (double x)
    {                                            /* compute acos(x) */
    return (_Asin(x, 1));
    }
```

图 7-22 asin.c

```
/* asin function */
#include <math.h>

double (asin)(double x)
    {                                            /* compute asin(x) */
    return (_Asin(x, 0));
    }
```

宏 DSIGN

正如我们看到的，函数 atan 仅仅提供了 atan2 所固有的功能的一个子集。那是因为 atan(y) 等价于 atan2(y,1.0)。顺便说一下，头文件 "xmath.h" 对宏 DSIGN 的定义如下：

```
define DSIGN(x) ( ( (unsigned short *) & (x) ) [D0] & _DSIGN)
```

该宏用来检查一个特殊编码（例如 Inf）的符号位，这在一个正常的表达式中可能不太好测试。在任何这样的特殊编码可能出现的地方，我都使用 DSIGN 来检测符号位。

atan2 首先检查它的参数是否为各种特殊编码。它接受任意定义了以原点为端点的半径向量的方向的一对数。（对 atan2(0,0) 的处理是有争议的，在专家建议的基础上，我选择了返回零。）。然后函数确定 _Atan 的两个参数 z 是缩减到区间 [0,1] 的正切参数。hex 把圆等分为 16 份：

图 7-23 atan.c

```
/* atan function */
#include "xmath.h"

double (atan) (double x)
    {                                               /* compute atan(x) */
    unsigned short hex;
    static const double piby2 = {1.57079632679489661923};

    switch (_Dtest(&x))
        {                                   /* test for special codes */
    case NAN:
        errno = EDOM;
        return (x);
    case INF:
        return (DSIGN(x) ? -piby2 : piby2);
    case 0:
        return (0.0);
    default:                                                /* finite */
        if (x < 0.0)
            x = -x, hex = 0x8;
        else
            hex = 0x0;
        if (1.0 < X)
            x = 1.0 / x, hex ^= 0x2;
        return (_Atan(x, hex));
        }
    }
```

- 如果 hex&0x8，取结果的负值。
- 如果 hex&0x4，z 的正切值加上 π/4。
- 如果 hex&0x2，π/4 减去 z 的正切值。
- 如果 hex&0x1，z 的正切值加 π/6。

只有 _Atan 设置最低有效位，来表明 z 的初始值大于 $2-3^{1/2}$（大约 0.268）。它用下面的表达式来代替 z：

```
(z*sqrt (3) -1) /sqrt (3) +z)
```

所有这些技巧都源自各种三角恒等式，这些三角恒等式用来缩减近似值的范围。

函数 _Atan 图 7-25 显示了文件 xatan.c，这个文件定义了函数 _Atan。因为它只能被函数 atan 和 atan2 调用，所以，它只检查其参数 x 是否需要减小到比 $2-3^{1/2}$ 小。如果减小后的参数的数值比 _Rteps._D 小，那么它就作为正切的近似值。否则，函数计算 Cody 和 Waite 提供的多项式的比例。函数加上表 a 中的一个元素来处理上面描述的所有的常数的加法和减法。

图 7-24
atan2.c

```
/* atan2 function */
#include "xmath.h"

double (atan2)(double y, double x)
    {                                               /* compute atan(y/x) */
    double z;
    const short errx = _Dtest(&x);
    const short erry = _Dtest(&y);
    unsigned short hex;

    if (errx <= 0 66 erry <= 0)
        {                                      /* x 6 y both finite or 0 */
        if (y < 0.0)
            y = -y, hex = 0x8;
        else
            hex = 0x0;
        if (x < 0.0)
            x = -x, hex ^= 0x6;
        if (x < y)
            z = x / y, hex ^= 0x2;
        else if (0.0 < x)
            z = y / x;
        else
            return (0.0) ;                                /* atan(0, 0) */
        }
    else if (errx == NAN || erry == NAN)
        {                                   /* return one of the NaNs */
        errno = EDOM;
        return (errx == NAN ? x : y);
        }
    else
        {                                          /* at least one INF */
        z = errx == erry ? 1.0 : 0.0;
        hex = DSIGN(y) ? 0x8 : 0x0;
        if (DSIGN(x))
            hex ^= 0x6;
        if (erry == INF)
            hex ^= 0x2;
        }
    return (_Atan(z, hex));
    }
```
□

函数 sqrt

最后一组函数是计算指数、对数和具体幂的函数。图 7-26 显示了文件 sqrt.c。函数 sqrt 计算它的参数 x 的平方根，或者 $x^{1/2}$。它使用 _Dunscale 把一个有限的正数 x 分为一个指数 e 和一个小数 f 两部分。参数值就是 $f*2^e$，其中 f 在区间 [0.5, 1.0] 内，那么它的平方根就是 $f^{1/2}*2^{e/2}$。

这个函数首先计算一个适合 $f^{1/2}$ 的一个二次最小二乘拟合。它然后 3 次应用牛顿法——除法和求均值——来获得需要的精度。注意函数是怎样把算法的最后两次迭代合并以提高它的性能的。

图 7-25 xatan.c

```
/* _Atan function */
#include "xmath.h"

/* coefficients, after Cody 6 Waite, Chapter 11 */
static const double a[8] = {
    0.0,
    0.52359877559829887308,
    1.57079632679489661923,
    1.04719755119659774615,
    1.57079632679489661923,
    2.09439510239319549231,
    3.14159265358979323846,
    2.61799387799149436538);
static const double p[4] = {
    -0.83758299368150059274e+0,
    -0.84946240351320683534e+1,
    -0.20505855195861651981e+2,
    -0.13688768894191926929e+2);
static const double q[5] = {
    0.10000000000000000000e+1,
    0.15024001160028576121e+2,
    0.59578436142597344465e+2,
    0.86157349597130242515e+2,
    0.41066306682575781263e+2);
static const double fold = {0.26794919243112270647);
static const double sqrt3 = {1.73205080756887729353);
static const double sqrt3ml = {0.73205080756887729353);

double _Atan(double x, unsigned short idx)
    {                               /* compute atan(x), 0 <= x <= 1.0 */
    i f (fold < x)
        {                                           /* 2-sqrt (3) < x */
        x = (((sqrt3ml * x - 0.5) - 0.5) + x) / (sqrt3 + x);
        idx |= 0x1;
        }
    if (X < -_Rteps._D || _Rteps._D < x)
        {                                 /* x*x worth computing */
        const double g = x * x;

        x += x * g / _Poly(g, q, 4)
            * (((p[0] * g + p[1]) * g + p[2]) * g + p[3]);
        }
    if (idx 6 0x2)
        X = -x;
    x += a[idx 6 07];
    return (idx 6 0x8 ? -x : x);
    }
```
□

图 7-26 sqrt.c

```
/* sqrt function */
#include <limits. h>
#include "xmath. h"

double (sqrt)(double x)
    {                                               /* compute sqrt(x) */
    short xexp;

    switch (_Dunscale(&xexp, &x))
        {                                   /* test for special codes */
    case NAN:
        errno = EDOM;
        return (x);
    case INF:
        if (DSIGN(x))
            {                                           /* -INF */
            errno = EDOM;
            return (_Nan._D);
            }
        else
            {                                           /* +INF */
            errno = ERANGE;
            return (_Inf._D);
            }
    case 0:
        return (0.0);
    default:                                            /* finite */
        if (x < 0.0)
                {                       /* sqrt undefined for reals */
                errno = EDOM;
                return (_Nan._D) ;
                }
            {                               /* 0 < x, compute sqrt(x) */
            double y;
            static const double sqrt2 = {1.41421356237309505};

            y = (-0.1984742 * x + 0.8804894) * x + 0.3176687;
            y = 0.5 * (y + x / y);
            y += x / y;
            x = 0.25 * y + x / y;
            if ((unsigned int)xexp & 1)
                x *= sqrt2, --xexp;
            _Dscale(&x, xexp / 2);
            return (x);
             }
            }
        }
```
□

7

图 7-27
xexp.c

```
/* _Exp function */
#include "xmath.h"

/* coefficients, after Cody & Waite, Chapter 6 */
static const double p[3] = {
    0.31555192765684646356e-4,
    0.75753180159422776666e-2,
    0.25000000000000000000e+0};
static const double q[4] = {
    0.75104028399870046114e-6,
    0.63121894374398503557e-3,
    0.56817302698551221787e-1,
    0.50000000000000000000e+0};
static const double c1 = {22713.0 / 32768.0};
static const double c2 = {1.4286068203094172321e-6};
static const double hugexp = {(double)HUGE_EXP};
static const double invln2 = {1.4426950408889634074};

short _Exp(double *px, short eoff)
    {                           /* compute e^(*px)*2^eoff, x finite */
    int neg;

    if (*px < 0)
        *px = -*px, neg = 1;
    else
        neg = 0;
    if (hugexp < *px)
        {                          /* certain underflow or overflow */
        *px = neg ? 0.0:_Inf._D;
        return (neg ? 0:INF);
        }
    else
        {                                    /* xexp won't overflow */
        double g = *px *invln2;
        short xexp = (short)(g + 0.5);

        g = (double)xexp;
        g = (*px - g * c1) - g * c2;
        if (-_Rteps._D < g && g < _Rteps._D)
            *px = 1.0;
        else
            {                                /* g*g worth computing */
            const double y = g * g;

            g *= (p[0] * y + P[1]) * y + p[2];
            *px= 0.5 + g / (((q[0] * y + q[1]) * y + q[2]) * y
                + q[3] - g);
            ++xexp;
            }
        if (neg)
            *px = 1.0 / *px, xexp = -xexp;
        return (_Dscale(px, eoff + xexp));
        }
    }
```
□

函数 _Exp

图 7-27 显示了文件 xexp.c，这个文件定义了函数 _Exp。有些函数需要计算一个有限参数的指数，或者 e^x。很多这样的函数实际上需要计算 $e^x/2$。在这种情况下，参数 eoff 是 -1。只有在 $e^x/2$ 发生溢出时结果才会溢出。

宏 HUGE_EXP

头文件 "xmath.h" 把宏 HUGE_EXP 定义为下面精心设计的值：

```
#define HUGE_EXP (int) (_DMAX * 900L / 1000)
```

这个值在所有已知的浮点数表示中，大到可以造成溢出。在以下的计算中，它也可以足够小但不至于造成整数溢出。因此，HUGE_EXP 对 _Exp 的那些实际上没有意义的参数提供了一个粗略的过滤。

这里用到技巧的地方是计算 ln(2) 除以 x，假设商为 y，然后求 2 的 y 次幂。你可以先去掉整数部分，计算 2^g，g 在区间 [–0.5,0.5] 内。然后用 _Dscale 函数在其结尾处把整数部分（加上 eoff）加上去。该函数也可以安全地处理所有的上溢和下溢。

通过这种方式减小参数，会遇到很多和前面讲过的减小 _Sin 和 tan 的参数相同的问题。这里的优势是可以选择扩展精度的常数 c1 和 c2 来为所有合理的参数值表示 1/ln(2)。

和以前一样，减小的参数通过和 _Rteps._D 比较来避免下溢和不必要的计算。多项式的比例系数还是来源于 Cody 和 Waite。近似值实际计算 $2^g/2$，因此需要对 xexp 进行修正。

函数 exp

图 7-28 显示了文件 exp.c。函数 exp 在调用带一个有限参数的 _Exp 之前检查它的参数是否为特殊编码。然后它检测返回值是否为零或者无穷大，来决定是否报告值域错误。

函数 cosh

图 7-29 显示了文件 cosh.c。函数 cosh 除了检查它的参数是否为特殊编码和调用 _Exp，基本上没什么其他事情了。因为函数的值不论是通过什么方式计算出来的，都取决于 exp(x)/2：

- 如果 x<_Xbig._D，那么函数值就是 (exp(x)+exp(-x))/2。这种形式去掉了第二个函数调用和一些算术运算。
- 否则，函数值是直接从 _Exp 获得的 exp(x)/2。如果 _Exp(x,-1) 发生了溢出，函数 cosh 也必须报告值域错误。

函数 sinh

图 7-30 显示了文件 sinh.c。函数 sinh 也是根据 _Exp 来计算定义域的大部分的值。但是和 cosh 不一样，它是一个奇函数。当参数的数值小于 1.0 时，常规的定义 (exp(x)-exp(-x))/2 就会丢失精度。在这个区间上，最好用多项式的一个比例值来求函数的近似值，这还是源于 Cody 和 Waite 的方法。和往常一样，如果 x 的数值比 _Rteps._D 小，参数值本身就是函数值的一个合适的近似值。

图 7-28
exp.c

```
/* exp function */
#include "xmath.h"

double (exp)(double x)
    {                                          /* compute exp(x) */
    switch (_Dtest(&x))
        {                                /* test for special codes */
    case NAN:
        errno = EDOM;
        return (x);
    case INF:
        errno = ERANGE;
        return (DSIGN(x) ? 0.0 : _Inf._D);
    case 0:
        return (1.0);
    default:                                         /* finite */
        if (0 <= _Exp(&x,0))
            errno = ERANGE;
        return (x);
        }
    }
```

图 7-29
cosh.c

```
/* cosh function */
#include "xmath.h"

double (cosh) (double x)
    {                                         /* compute cosh(x) */
    switch (_Dtest(&x))
        {                                /* test for special codes */
    case NAN:
        errno = EDOM;
        return (x);
    case INF:
        errno = ERANGE;
        return (_Inf._D);
    case 0:
        return (1.0);
    default:                                         /* finite */
        if (x < 0.0)
            x = -x;
        if (0 <= _Exp(&x, -1))
                errno = ERANGE;                     /* x large */
        else if (x < _xbig._D)
            x += 0.25 / x;
        return (x);
        }
    }
```

图 7-30 sinh.c

```
/* sinh function */
#include "xmath.h"

/* coefficients, after Cody & Waite, Chapter 12 */
static const double p[4] = {
    -0.78966127417357099479e+0,
    -0.16375798202630751372e+3,
    -0.11563521196851768270e+5,
    -0.35181283430177117881e+6};
static const double q[4] = {
    1.0,
    -0.27773523119650701667e+3,
     0.36162723109421836460e+5,
    -0.21108770058106271242e+7};

double (sinh)(double x)
    {                                       /* compute sinh(x) */
    switch (_Dtest(&x))
        {                               /* test for special codes */
    case NAN:
        errno = EDOM;
        return (x);
    case INF:
        errno = ERANGE;
        return (DSIGN(x) ? -_Inf._D : _Inf._D);
    case 0:
        return (0.0);
    default:                                        /* finite */
         {                                /* compute sinh(finite) */
        short neg;

        if (x < 0.0)
            x = -x, neg = 1;
        else
            neg = 0;
        if (x < _Rteps._D)
            ;                                       /* x tiny */
        else if (x < 1.0)
            {                                      /* |x| < 1 */
            const double y = x * x;

            x += x * y
                * (((p[0] * y + p[1]) * y + p[2]) * y + p[3])
                / (((q[0] * y + q[1]) * y + q[2]) * y + q[3]);
            }
        else if (0 <= _Exp(&x, -1))
            errno = ERANGE;                        /* x large */
        else if (x < _xbig._D)
            x -= 0.25 / x;
        return (neg ? -x : x);
         }
        }
    }
```
□

函数 tanh 图 7-31 显示了文件 tanh.c。函数 tanh 在很多方面和 sinh 都很相似。有一点不同就是它不会溢出。当函数的参数 x 的数值增大时，函数值可以达到±1.0。（这个函数可以和 cosh 和 sinh 一样把 x 和 _Rteps._D 相比较。然而，返回给 _Exp 的溢出编码可当作一个有效的提示。）其他的区别就在函数选择改变比例多项式近似法的地方。这里使用的和 Cody 和 Waite 的不同，小于 ln(3)/2（大约 0.549）的 x 的数值是精确的。

函数 log 图 7-32 显示了文件 log.c。它通过调用 _Log(x,0) 来计算 log(x)。很自然地，头文件 <math.h> 为这个函数提供了一个屏蔽宏。这看起来没什么用，但它却是为 log10（下面将要讲到）提供屏蔽宏的最安全的方式。

函数 _Log 图 7-33 显示了文件 xlog.c，这个文件定义了函数 _Log。这个函数使用以前在 _Exp 中使用的技巧来计算自然对数，只不过反过来了。这种方法是使用 _Dunscale 函数去掉二元指数 e，剩下小数部分 f。参数值是 $f*2^e$，在这里 f 在区间 [0.5,1.0] 内。可以先计算 $f*2^e$ 的以 2 为底的对数得到 $\log_2(f)+e$。然后将其乘以 ln(2) 得到最终结果。

这种方法要求一些很精细的设计。Cody 和 Waite 提出的近似法要求 f 在区间 $[0.5^{1/2}, 2.0^{1/2}]$ 内。如果 f（实际的 x）太小，就要把它加倍然后修正 e(xexp)。同时还要引入一个新的变量 $z=(f-1)/(f+1)$，把两种操作结合起来并消除那些可能会造成精度丢失的步骤会更好。近似值仍然是多项式的另一个比例。注意，它实际上计算的是自然对数。所以在求和之前，只需要改变 xexp 的大小。

求和时一定要非常小心，至少对接近零的对数要如此，这是 _Exp 中的参数缩减所产生的另一个问题。这两个函数都使用和 ln(2) 相同的扩展精度表示。这里，在最后的结果中，较小的部分先和尽可能多的转换常数的低位拼接起来，然后才是较大的那一部分。

log10 图 7-34 显示了文件 log10.c。它通过调用 _Log 并将其结果乘以 $\log_{10}(e)$ 来计算以 10 为底的对数。发生在 _Log 内部的乘法仅仅是为了得到一个有限的结果。

函数 pow 图 7-35 显示了文件 pow.c。计算 x 的 y 次幂的函数 pow，很自然地成了所有数学函数中最复杂的一个。它必须处理很多特殊情况，也必须设法为各种参数值计算出精确的结果。

到现在为止，你应该意识到计算 exp(y*log(x)) 中的危险了。可以简单地把表示 x 的指数的对数偏移若干个小数位作为它的整数部分。乘以 y 可能会使情况变得更糟。指数会使这些整数位再次回到指数位位置，但是损害已经发生了。除非可以使用中间计算来提高精度，否则在这个过程中必须丢

图 7-31 tanh.c

```
/* tanh function */
#include "xmath.h"

/* coefficients, after Cody & Waite, Chapter 13 */
static const double p[3] = {
    -0.96437492777225469787e+0,
    -0.99225929672236083313e+2,
    -0.16134119023996228053e+4};
static const double q[4] = {
    0.10000000000000000000e+1,
    0.11274474380534949335e+3,
    0.22337720718962312926e+4,
    0.48402357071988688686e+4};
static const double ln3by2 = {0.54930614433405484570};

double (tanh)(double x)
    {                                        /* compute tanh(x) */
    switch (_Dtest(&x) )
        {                                /* test for special codes */
    case NAN:
        errno = EDOM;
        return (x);
    case INF:
        return (DSIGN(x) ? -1.0 : 1.0);
    case 0:
        return (0.0);
    default:                                          /* finite */
         {                             /* compute tanh(finite) */
        short neg;

        if (x < 0.0)
            x = -x, neg = 1;
        else
            neg = 0;
        if (x < _Rteps._D)
            ;                                         /* x tiny */
        else if (x < ln3by2)
            {                                 /* |x| < ln(3)/2 */
            const double g = x * x;

            x += x * g * ((p[0] * g + p[1]) * g + p[2])
                / (((q[0] * g + q[1]) * g + q[2]) * g + q[3]);
            }
        else if (_Exp(&x, 0) < 0)
            x = 1.0 - 2.0 / (x * x + 1.0);
        else
            x = 1.0;                                 /* x large */
        return (neg ? -x : x);
         }
        }
    }
```
□

图 7-32 log.c

```
/* log function */
#include <math.h>

double (log)(double x)
    {                                              /* compute ln(x) */
    return (_Log(x, 0));
    }
```

图 7-33 xlog.c

```
/* _Log function */
#include "xmath.h"

/* coefficients, after Cody & Waite, Chapter 5 */
static const double p[3] = {
    -0.78956112887491257267e+0,
     0.16383943563021534222e+2,
    -0.64124943423745581147e+2};
static const double q[3] = {
    -0.35667977739034646171e+2,
     0.31203222091924532844e+3,
    -0.76949932108494879777e+3};
static const double c1 = {22713.0 / 32768.0};
static const double c2 = {1.428606820309417232e-6};
static const double loge = 0.43429448190325182765;
static const double rthalf = (0.70710678118654752440);

double _Log(double x, int decflag)
    {                                              /* compute ln(x) */
    short xexp;

    switch (_Dunscale(&xexp, &x))
        {                                /* test for special codes */
    case NAN:
        errno = EDOM;
        return (x);
    case INF:
        if (DSIGN(x))
            {                                          /* -INF */
            errno = EDOM;
            return (_Nan._D);
            }
        else
            {                                          /* INF */
            errno = ERANGE;
            return (_Inf._D);
            }
    case 0:
        errno = ERANGE;
        return (-_Inf._D);
    default:                                           /* finite */
        if (x < 0.0)
            {                           /* ln(negative) undefined */
            errno = EDOM;
            return (_Nan._D);
            }
```

图 7-33（续）

```
            else
                {                                       /* 1/2 <= x < 1 */
                double z = x - 0.5;
                double w;

                if (rthalf < x)
                    z = (z - 0.5) / (x * 0.5 + 0.5);
                else
                    {                                   /* x <= sqrt(1/2) */
                    --xexp;
                    z /= (Z * 0.5 + 0.5);
                    }
                w = z * z;
                z += z * w * ((p[0] * w + p[1]) * w + p[2])
                    / (((w + q[0]) * w + q[1]) * w + q[2]);
                if (xexp != 0)
                    {                   /* form z += ln2 * xexp safely */
                    const double xn = (double)xexp;

                    z = (xn * c2 + z) + xn * c1;
                    }
                return (decflag ? loge * z : z);
                }
            }
        }
```

图 7-34 log10.c

```
/* log10 function */
#include <math.h>

double (log10)(double x)
    {                                           /* compute log10(x) */
    return (_Log(x, 1));
    }
```

失某些有效位。pow 的这种实现有效地保持了这种扩展精度，而无需借助比 *double* 有更多位数的数据类型。

这个函数的前半部分简单地对参数值的各种组合进行整理。或者 x 是零，或者至少有一个参数是 Inf 或者 NaN。我已经设计了一种具有启发作用的方法来把这些情况列成表格。以后我们必须跟踪这段代码来看看它是怎样处理各种各样的组合的，这里我再一次采纳了专业人士的建议。C 标准对此并没有提供什么指导。

顺便说一下，你可能会注意到函数是如何调用 _Dint(&y,-1) 来确定存储在 *double* 类型的 y 中的整值是偶数还是奇数的。此时，_Dint 清空了 y 的整数部分的最低有效位。如果它清空的位初始时是非零的，它就返回一个负的 FINITE 编码。在函数 pow 的后面的部分也会有一个相似的检测。

图 7-35 pow.c

```
/* pow function */
#include "xmath.h"

double (pow)(double x, double y)
    {                                           /* compute x^y */
    double yi = y;
    double yx, z;
    short n, xexp, zexp;
    short neg = 0;
    short errx = _Dunscale(&xexp, &x);
    const short erry = _Dint(&yi, 0);
    static const short shuge = {HUGE_EXP};
    static const double dhuge = {(double)HUGE_EXP};
    static const double ln2 = {0.69314718055994530942};
    static const double rthalf = {0.70710678118654752440};

    if (0 <= errx || 0 < erry)
        {                     /* x == 0, INF, NAN; y == INF, NAN */
        z = _Nan._D;
        if (errx == NAN || erry == NAN)
            z = errx == NAN ? x : y, errx = NAN;
        else if (erry == INF)
            if (errx == INF)                        /* INF^INF */
                errx = INF;
            else                          /* 0^INF, finite^INF */
                errx = xexp <= 0 ? (DSIGN(y) ? INF : 0)
                    : xexp == 1 && (x == 0.5 || x == -0.5) ? NAN
                    : (DSIGN(y) ? 0 : INF);
        else if (y == 0.0)
            return (1.0);                  /* x^0, x not a NaN */
        else if (errx == INF)
            {           /* INF^finite (NB: erry tests y fraction) */
            errx = y < 0.0 ? 0 : INF;
            neg = DSIGN(x) && erry == 0 && _Dint(&y, -1) < 0;
            }
        else                                       /* 0^finite */
            errx = y < 0.0 ? INF : 0;
        if (errx == 0)
            return (0.0);
        else if (errx == INF)
            {                           /* return -INF or INF */
            errno = ERANGE;
            return (neg ? -_Inf._D : _Inf._D);
            }
        else
            {                                   /* return NaN */
            errno = EDOM;
            return (z);
            }
        }
    if (y == 0.0)
        return (1.0);
    if (0.0 < x)
        neg = 0;
```

图 7-35
（续）

```
else if (erry < 0)
    {                                   /* negative^fractional */
    errno = EDOM;
    return (_Nan._D);
    }
else
    x = -x, neg = _Dint(&yi, -1) < 0;
if (X < rthalf)
    x *= 2.0, --xexp;           /* -sqrt(.5) <= x <= sqrt(.5) */
n = 0, yx = 0.0;
if (y <= -dhuge)
    zexp = xexp < 0 ? shuge : xexp == 0 ? 0 : -shuge;
else if (dhuge <= y)
    zexp = xexp < 0 ? -shuge : xexp == 0 ? 0 : shuge;
else
    {                          /* y*log2(x) may be reasonable */
    double dexp = (double)xexp;
    long zl = (long)(yx = y * dexp);

    if (zl != 0)
        {                      /* form yx = y*xexp-zl carefully */
        yx = y, _Dint(&yx, 16);
        yx = (yx * dexp - (double)zl) + (y - yx) * dexp;
        }
    yx *= ln2;
    zexp = zl <= -shuge ? -shuge : zl < shuge ? zl : shuge;
    if ((n = (short)y) < -SAFE_EXP || SAFE_EXP < n)
        n = 0;
    }
 {                      /* compute z = xfrac^n * 2^yx * 2^zexp */
z = 1.0;
if (x != 1.0)
    {                                          /* z *= xfrac^n */
    if ((yi = y - (double)n) != 0.0)
        yx += log(x) * yi;
    if (n < 0)
        n = -n;
    for (yi = x; ; yi *= yi)
        {                                    /* scale by x^2^n */
        if (n & 1)
            z *= yi;
        if ((n >>= 1) == 0)
            break;
        }
    if (y < 0.0)
        z = 1.0 / z;
    }
if (yx != 0.0)                                   /* z *= 2^yx */
    z = _Exp(&yx, 0) < 0 ? z * yx : yx;
if (0 <= _Dscale(&z, zexp))                    /* z *= z^zexp */
    errno = ERANGE;                  /* underflow or overflow */
return (neg ? -z : z);
  }
}
```
□

宏 SAFE_EXP　函数的后半部分计算 x 和 y 为有限值时的 x^y。开始的时候，它把 x 重写为 $f*2^e$，这里 f 在区间 $[0.5^{1/2}, 2.0^{1/2}]$ 内。如果 N 是最大的可表示的 *double* 类型的指数的数值，那么，就可以计算 f 的 N 次幂而不用担心溢出。结果的指数的值不会超过 $N/2$。头文件 "xmath.h" 把宏 SAFE_EXP 定义为：

```
#define SAFE-EXP (-DMAX>>1)
```

pow 使用这个值仅仅是为了这样的一个检查。

也可以把 x^y 重写为 f^y*2^{e*y}，然后把乘积 $e*y$ 分成一个整数加一个小数，或者 $n+g$，这里 g 在区间 $(-1,1)$ 的范围内。现在可以把函数重新写为：

$$x^y=f^n*(f^{y-n}*2^g)*2n$$

我故意将中间的两项组成一组，这样就把问题简化为了3项的乘积：

- f^n 是一个用 f 乘以它自己 $|n|$ 次的循环。如果 n 是负的，结果就被1除。只要 $|n|$ 比 SAFE_EXP 小，结果就不会上溢或者下溢，前面已经说明了其中的原因。
- $(f^{y-n}*2^g)$ 可以作为 $(y-n)*ln(f)+g*ln(2)$ 的指数来求值。该求和算式中的两项都很小，所以在加法或者求幂的过程中不会出现严重的精度丢失。当然 $|n|$ 大于 SAFE_EXP 的情况除外。在这种情况下，函数把 n（在代码中也表示为 n）设置为零并且不再关心精度的问题。不管 y（yi）有多大，结果都不会发生溢出。如果指数没有溢出，那么，无论如何，最后的结果大概就由这个项来决定了。
- 2^n 是对 _Dscale 的一次简单的调用。

这个计算过程最复杂的地方就在于避免上溢和下溢，其次就是把 $e*y$ 分为 n 和 g 的和。注意这里对 _Dint 的使用是另一种方式。该方式使用 yx 来提高精度，从而让你在 y 中保留额外的16位精度。这样就弥补了划分期间精度的丢失。这个实现所支持的最大的浮点指数被假定不超过14位，因此这种划分应该在所有可表示的值的范围内都很安全。

其他函数　为了保持完整性，我给出了两个没有被 <math.h> 中的其他函数使用的函数。但其他标准头文件中声明的函数用到了它们。这两个函数需要包含 "xmath.h"。看起来把这两个函数放在这里是最好的选择。

函数 _Dtento　图7-37显示了文件 xdtento.c，这个文件定义了函数 _Dtento。该函数用10的 n 次方来乘以 *double* 类型的值 x。在这个过程中，它避免了浮点上溢和下溢。注意内部函数 dmul 中对 _Dunscale 和 _Dscale 的使用。无论 _Dscale 中发生了任何潜在的上溢或者下溢，它都会安全地对其进行处理。函数 _Dtento 假设参数 x 是零或者有限大。

函数 _Ldunscale 图 7-36 显示文件 xldunsca.c，它定义了函数 _Ldunscale。该函数做着和 _Dunscale 相同的工作，只不过它的参数是 *long double* 类型的，而 _Dunscale 的参数是 *double* 类型的。事实上，如果这两种浮点类型的表示相同，那么它的工作就和 _Dunscale 的完全相同了。只有当 _DLONG 为非零时，_Ldunscale 才处理 10 字节的 IEEE 754 扩展精度格式。

头文件 "xmath.h" 图 7-38 显示了文件 xmath.h。到现在为止，应该已经介绍了其中的所有奥秘了。在这里，为了保持完整性，我把它完整地显示了出来。

7.5 <math.h> 的测试

测试数学函数是一个严肃的工作，因为即使是那些半数值函数也有很多出错的可能，其他的数学函数则要求在技术上多花费功夫以保证其正确性。这就是我使用 elefunt 测试来验证三角函数、指数、对数和特殊的幂函数的原因。

在使用 IEEE 754 浮点算术的 Sun 3 工作站下，这些测试报告的最坏情况的错误是一个小于两位的精确度的丢失。均方根错误一般情况下比两位要好得多。

paranoia 测试偶然也报告一个小于两位的错误。(这里容易出错的是 sqrt 和对某些极值的格式化输入输出函数。) 4.6 中讲述了如何才能获得 paranoia 程序。

我也提供了测试 <math.h> 中声明的所有函数的一个测试集合。每一个函数有几个测试实例，足以验证它是健全的。然而，要验证 <math.h> 中声明的所有的函数，总共需要很多的测试。所以，我把这些测试分到了 3 个文件中，每一个文件对应 3 组函数中的每一组。

程序 tmath1.c 图 7-39 显示了文件 tmath1.c。它测试了宏 HUGE_VAL 和所有的半数值函数。某些测试可以产生精确的结果，其他的可能会产生一些小错误。对于后者来说，函数 approx 检测出结果丢失了小于两位的精度。这个程序也显示了打印函数对 HUGE_VAL 的输出。

如果这个库运行在一个支持 Inf 和 NaN 两种特殊编码的计算机体系结构下，这个程序显示如下输出：

```
SUCCESS testing <math.h>, part 1
```

程序 tmath2.c 图 7-40 显示了文件 tmath2.c。它测试了所有的三角函数在角度为 π/4 的各个倍数上的表现。这些通常是检测精度丢失或者确定结果的符号错误方面的关键的角度值。如果所有的测试都通过了，程序显示如下信息：

```
SUCCESS testing <math .h>, part 2
```

图 7-36 xldunsca.c

```
/* _Ldunscale function -- IEEE 754 version */
#include "xmath.h"

#if _DLONG                                          /* 10-byte IEEE format */
#define _LMASK 0x7fff
#define _LMAX 0x7fff
#define _LSIGN 0x8000
#if _D0==3
#define _L0   4                                     /* little-endian order */
#define _L1   3
#define _L2   2
#define _L3   1
#define _L4   0
#else
#define _L0   0                                        /* big-endian order */
#define _L1   1
#define _L2   2
#define _L3   3
#define _L4   4
#endif

static short dnorm(unsigned short *ps)
    {                               /* normalize long double fraction */
    short xchar;

    for (xchar = 0; ps[_L1] == 0; xchar -= 16)
        {                                          /* shift left by 16 */
        ps[_L1] = ps[_L2], ps[_L2] = ps[_L3];
        ps[_L3] = ps[_L4], ps[_L4] = 0;
        }
    for (; ps[_L1] < 1U<<_LOFF; --xchar)
        {                                           /* shift left by 1 */
        ps[_L1] = ps[_L1] << 1 | ps[_L2] >> 15;
        ps[_L2] = ps[_L2] << 1 | ps[_L3] >> 15;
        ps[_L3] = ps[_L3] << 1 | ps[_L4] >> 15;
        ps[_L4] <<= 1;
        }
    return (xchar);
    }

short _Ldunscale(short *pex, long double *px)
    {                       /* separate *px to |frac| < 1/2 and 2^*pex */
    unsigned short *ps = (unsigned short *)px;
    short xchar = ps[_L0] & _LMASK;

    if (xchar == _LMAX)
        {                                               /* NaN or INF * /
        *pex = 0;
        return (ps[_L1] & 0x7fff || ps[_L2]
            || ps[_L3] || ps[_L4] ? NAN : INF);
        }
```

图 7-36
（续）

```
    else if (ps[_L1] == 0 && ps[_L2] == 0
        && ps[_L3] == 0 && ps[_L4] == 0)
        {                                               /* zero */
        *pex = 0;
        return (0);
        }
    else
        {                               /* finite, reduce to [1/2, 1) */
        xchar += dnorm(ps);
        ps[_L0] = ps[_L0] & _LSIGN | _LBIAS;
        *pex = xchar - _LBIAS;
        return (FINITE);
        }
    }
#else                                   /* long double same as double */
short _Ldunscale(short *pex, long double *px)
    {                           /* separate *px to |frac| < 1/2 and 2^*pex */
    unsigned short *ps = (unsigned short *)px;
    short xchar = (ps[_D0] & _DMASK) >> _DOFF;

    if (xchar == _DMAX)
        {                                           /* NaN or INF */
        *pex = 0;
        return (ps[_D0] & _DFRAC || ps[_D1]
            || ps[_D2] || ps[_D3] ? NAN : INF);
        }
    else if (0 < xchar || (xchar = _Dnorm(ps)) != 0)
        {                               /* finite, reduce to [1/2, 1) */
        ps[_D0] = ps[_D0] & ~_DMASK | _DBIAS << _DOFF;
        *pex = xchar - _DBIAS;
        return (FINITE);
        }
    else
        {                                               /* zero */
        *pex = 0;
        return (0);
        }
    }
#endif
```

程序 tmath3.c

图 7-41 显示了文件 tmath3.c。它测试了所有的指数、对数和特殊的幂函数的一些明显的属性。注意，其中的一两个测试必须生成精确的结果。如果所有的测试都通过了，程序显示如下信息：

```
SUCCESS testing <math.h>, part 3
```

我很惭愧地说，这些简单的测试出现了很多错误。有一些错误在我第一次编写和调试数学函数的时候就产生了。当我引入各种“改进”的时候，又有些更加令人难堪的错误出现了。因此，我学会了在任何修改之后认真地重新运行它们。

图 7-37
xdtento.c

```
/* _Dtento function -- IEEE 754 version */
#include <errno.h>
#include <float.h>
#include "xmath.h"

        /* macros */
#define NPOWS  (sizeof pows / sizeof pows[0] - 1)
        /* static data */
static const double pows [] = {
    1e1, 1e2, 1e4, 1e8, 1e16, 1e32,
#if 0x100 < _DBIAS                        /* assume IEEE 754 8-byte */
    1e64, 1e128, 1e256,
#endif
    };
static const size_t npows = {NPOWS};

static short dmul(double *px, double y)
    {                           /* multiply y by *px with checking */
    short xexp;

    _Dunscale(&xexp, px);
    *px *= y;
    return (_Dscale(px, xexp));
    }

double _Dtento(double x, short n)
    {                                           /* compute x * 10**n */
    double factor;
    short errx;
    size_t i;

    if (n == 0 || x == 0.0)
        return (x);
    factor = 1.0;
    if (n < 0)
        {                                               /* scale down */
        unsigned int nu = -(unsigned int)n;

        for (i = 0; 0 < nu && i < npows; nu >>= 1, ++i)
            if (nu & 1)
                factor *= pows[i];
        errx = dmul(&x, 1.0 / factor);
        if (errx < 0 && 0 < nu)
            for (factor = 1.0 / pows[npows]; 0 < nu; --nu)
                if (0 <= (errx = dmul(&x, factor)))
                    break;

        }
    else if (0 < n)
        {                                                 /* scale up */
        for (i = 0; 0 < n && i < npows; n >>= 1, ++i)
            if (n & 1)
                factor *= pows[i];
```

图 7-37
（续）

```
        errx = dmul(&x, factor);
        if (errx < 0 && 0 < n)
            for (factor = pows[npows]; 0 < n; --n)
                if (0 <= (errx = dmul(&x, factor)))
                    break;
        }
    if (0 <= errx)
        errno = ERANGE;
    return (x);
    }
```

图 7-38
xmath.h

```
/* xmath.h internal header -- IEEE 754 version */
#include <errno.h>
#include <math.h>
#include <stddef.h>
#ifndef _YVALS
#include <yvals.h>
#endif
        /* IEEE 754 properties */
#define _DFRAC ((1<<_DOFF)-1)
#define _DMASK (0x7fff&~_DFRAC)
#define _DMAX ((1<<(15-_DOFF))-1)
#define _DNAN (0x8000|_DMAX<<_DOFF|1<<(_DOFF-1))
#define _DSIGN 0x8000
#define DSIGN(x) (((unsigned short *)&(x))[_D0] & _DSIGN)
#define HUGE_EXP (int)(_DMAX * 900L / 1000)
#define HUGE_RAD  3.14e30
#define SAFE_EXP (_DMAX>>1)
        /* word offsets within double */
#if _D0==3
#define _D1   2                               /* little-endian order */
#define _D2   1
#define _D3   0
#else
#define _D1   1                                  /* big-endian order */
#define _D2   2
#define _D3   3
#endif
        /* return values for _D functions */
#define FINITE   -1
#define INF   1
#define NAN   2
        /* declarations */
double _Atan(double, unsigned short);
short _Dint(double *, short);
short _Dnorm(unsigned short *);
short _Dscale(double *, short);
double _Dtento(double, short);
short _Dtest(double *);
short _Dunscale(short *, double *);
short _Exp(double *, short);
short _Ldunscale(short *, long double *);
double _Poly(double, const double *, int);
extern _Dconst _Inf, _Nan, _Rteps, _Xbig;
```

图 7-39
tmath1.c

```
/* test math functions    -- part 1 */
#include <assert.h>
#include <float.h>
#include <math.h>
#include <stdio.h>

static double eps;

static int approx(double d1, double d2)
    {                              /* test for approximate equality */
    if (d2 != 0)
        return (fabs((d2 - d1) / d2) < eps);
    else
        return (fabs(d1) < eps);
    }

int main ()
    {                         /* test basic workings of math functions */
    double huge_val, x;
    int xexp;

    huge_val = HUGE_VAL;
    eps = DBL_EPSILON * 4.0;
    assert(ceil(-5.1) == -5.0);
    assert(ceil(-5.0) == -5.0);
    assert(ceil(-4.9) == -4.0);
    assert(ceil(0.0) == 0.0);
    assert(ceil(4.9) == 5.0);
    assert(ceil(5.0) == 5.0);
    assert(ceil(5.1) == 6.0);
    assert(fabs(-5.0) == 5.0);
    assert(fabs(0.0) == 0.0);
    assert(fabs(5.0) == 5.0);
    assert(floor(-5.1) == -6.0);
    assert(floor(-5.0) == -5.0);
    assert(floor(-4.9) == -5.0);
    assert(floor(0.0) == 0.0);
    assert(floor(4.9) == 4.0);
    assert(floor(5.0) == 5.0);
    assert( floor(5.1) == 5.0);
    assert(fmod(-7.0, 3.0) == -1.0);
    assert(fmod(-3.0, 3.0) == 0.0);
    assert(fmod(-2.0, 3.0) == -2.0);
    assert(fmod(0.0, 3.0) == 0.0);
    assert(fmod(2.0, 3.0) == 2.0);
    assert(fmod(3.0, 3.0) == 0.0);
    assert(fmod(7.0, 3.0) == 1.0);
    assert(approx(frexp(-3.0, &xexp), -0.75) && xexp == 2);
    assert(approx(frexp(-0.5, &xexp), -0.5) && xexp == 0);
    assert(frexp(0.0, &xexp) == 0.0 && xexp == 0);
    assert(approx(frexp(0.33, &xexp), 0.66) && xexp == -1) ;
    assert(approx(frexp(0.66, &xexp), 0.66) && xexp == 0);
    assert(approx(frexp(96.0, &xexp), 0.75) && xexp == 7);
    assert(ldexp(-3.0, 4) == -48.0);
    assert(idexp(-0.5, 0) == -0.5);
```

图 7-39（续）

```
    assert(ldexp(0.0, 36) == 0.0);
    assert(approx(ldexp(0.66, -1), 0.33));
    assert(ldexp(96, -3) == 12.0);
    assert(approx(modf(-11.7, &x), -11.7 + 11.0)
        && x == -11.0);
    assert(modf(-0.5, &x) == -0.5 && x == 0.0);
    assert(modf(0.0, &x) == 0.0 && x == 0.0);
    assert(modf(0.6, &x) == 0.6&& x == 0.0);
    assert(modf(12.0, &x) == 0.0&& x == 12.0);
    printf("HUGE_VAL prints as %.16e\n", huge_val);
    puts("SUCCESS testing <math.h>, part 1");
    return(0);
    }
```

□

7.6 参考文献

William J. Cody, Jr.and William Waite, *Software Manual For the Elementary Functions* (Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1980). 在编写稳定的和高精度的数学函数方面，这是一本非常好的参考手册。它是本章中很多函数的近似值的来源。

John F. Hart、E.W. Cheney、Charles L. Lawson、Hans J. Maehly、Charles K. Mesztenyi、John R. Rice、Henry G. Thacher、Jr. 和 Christoph Witzgall,*Computer Approximations* (Malabar, Florida: Robert E. Krieger Publishing Company, 1978). 这本书中有几章是关于数值逼近的艺术和科学的，但是它大部分讲的都是它的广泛的系数表。对所有常用的数学函数，你都可能会找到一个你需要的精度的近似值。

elefunt 是一个可移植的 FORTRAN 程序集，这些程序在提供 FORTRAN 编译器的情况下，可以测试基本函数程序。它们非常完整。这些程序是 William J.Cody 用 FORTRAN 编写的，在 Cody 和 Waite 的书中有详细的讲解。你可以用以下请求来向 netlib@research.att.com 发邮件：

```
send index from elefunt
```

7.7 习题

7.1 确定你的 C 翻译器的浮点数表示。可以修改 <yvals.h> 中的参数来适应你的翻译器吗？如果可以，就实现它；否则，修改原语来适应它。

7.2 编写函数 double hypot(double, double)，这个函数计算它的参数的平方和的平方根（如果把直角三角形的两个直角边作为参数，函数的计算结果就是它的斜边）。用下面的表达式来测试它：

```
hypot (0.7 * DBL_MAX, 0.7 * DBL_MAX);
hypot (DBL_MAX, 1.0);
hypot (1.0, DBL_MAX);
hypot(3.0, 4.0);
```

图 7-40、tmath2.c

```
/* test math functions -- part 2 */
#include <assert.h>
#include <float.h>
#include <math.h>
#include <stdio.h>

        /* static data */
static double eps;

static int approx(double d1, double d2)
    {                           /* test for approximate equality */
return ((d2 ? fabs((d2 - d1) / d2) : fabs(d1)) < eps);
    }

int main()
    {                       /* test basic workings of math functions */
    double x;
    int xexp;
    static double piby4 = {0.78539816339744830962};
    static double rthalf = {0.70710678118654752440};

    eps = DBL_EPSILON * 4.0;
    assert(approx(acos(-1.0), 4.0 * piby4));
    assert(approx(acos(-rthalf), 3.0 * piby4));
    assert(approx(acos(0.0), 2.0 * piby4));
    assert(approx(acos(rthalf), piby4));
    assert(approx(acos(1.0), 0.0));
    assert(approx(asin(-1.0), -2.0 * piby4));
    assert(approx(asin(-rthalf), -piby4));
    assert(approx(asin(0.0), 0.0));
    assert(approx(asin(rthalf), piby4));
    assert(approx(asin(1.0), 2.0 * piby4));
    assert(approx(atan(-DBL_MAX), -2.0 * piby4));
    assert(approx(atan(-1.0), -piby4));
    assert(approx(atan(0.0), 0.0));
    assert(approx(atan(1.0), piby4));
    assert(approx(atan(DBL_MAX), 2.0 * piby4));
    assert(approx(atan2(-1.0, -1.0), -3.0 * piby4));
    assert(approx(atan2(-1.0, 0.0), -2.0 * piby4));
    assert(approx(atan2(-1.0, 1.0), -piby4));
    assert(approx(atan2(0.0, 1.0), 0.0));
    assert(approx(atan2(1.0, 1.0), piby4));
    assert(approx(atan2(1.0, 0.0), 2.0 * piby4));
    assert(approx(atan2(1.0, -1.0), 3.0 * piby4));
    assert(approx(atan2(0.0, -1.0), 4.0 * piby4)
        || approx(atan2(0.0, -1.0), -4.0 * piby4));
    assert(approx(cos(-3.0 * piby4), -rthalf));
    assert(approx(cos(-2.0 * piby4), 0.0));
    assert(approx(cos(-piby4), rthalf));
    assert(approx(cos(0.0), 1.0));
    assert(approx(cos(piby4), rthalf));
    assert(approx(cos(2.0 * piby4), 0.0));
    assert(approx(cos(3.0 * piby4), -rthalf));
    assert(approx(cos(4.0 * piby4), -1.0));
    assert(approx(sin(-3.0 * piby4), -rthalf));
```

图 7-40（续）

```
assert(approx(sin(-2.0 * piby4). -1.0));
assert(approx(sin(-piby4). -rthalf));
assert(approx(sin(0.0). 0.0));
assert(approx(sin(piby4). rthalf));
assert(approx(sin(2.0 *piby4), 1.0));
assert(approx(sin(3.0 *piby4), rthalf));
assert(approx(sin(4.0 *piby4). 0.0));
assert(approx(tan(-3.0 *piby4). 1.0));
assert(approx(tan(-piby4), -1.0));
assert(approx(tan(0.0). 0.0));
assert(approx(tan(piby4). 1.0));
assert(approx(tan(3.0 * piby4), -1.0));
puts("SUCCESS testing <math.h>, part 2");
return (0);
}
```
□

7.3 编写执行复数算术的函数。每一个复数值 $x+i*y$ 可以用一有序对 (x, y) 来表示。至少要提供比较、减法、加法、除法、乘法、求模和定相几种功能。也要提供可以在现有的浮点类型和复数类型之间进行转换的函数。你可以利用任何现有的函数吗？还需要其他的什么函数？

7.4 修改 <math.h> 中的原语，来消除 NaN、Inf 和 -Inf 这些特殊编码。在任何可能的地方用 "xmath.h" 中的宏来代替原语。这些对 C 标准库中的函数的大小有什么影响？对执行时间呢？

7.5 **[难]** 编写所有的可以接受 *float* 类型的参数和产生 *float* 类型的结果的数学函数版本。在每一个现有的函数名后面加上 f 作为新的函数名。怎样测试这些函数？

7.6 **[难]** 编写所有的可以接受 *long double* 类型参数并产生 *long double* 类型结果的数学函数的版本。在每一个现有的函数名后面加上 l 作为新的函数名。怎样测试这些函数？

7.7 **[难]** 编写所有的可以接受复数参数并产生复数结果的数学函数的版本。在每一个现有的函数名前面加上 c 作为新的函数名。怎样测试这些函数？

7.8 **[很难]** 通过比较一个很大的代码集，确定是否有些数学函数值得编写为内联代码。修改一个 C 编译器来实现它，对结果进行测量。

图 7-41 tmath3.c

```
/* test math functions -- part 3 */
#include <assert.h>
#include <float.h>
#include <math.h>
#include <stdio.h>

static double eps;

static int approx(double d1, double d2)
    {                               /* test for approximate equality */
    return ((d2 ? fabs((d2 - d1) / d2) : fabs(d1)) < eps);
    }

int main()
    {                       /* test basic workings of math functions */
    double x;
    int xexp;
    static double e = {2.71828182845904523536};
    static double ln2 = {0.69314718055994530942};
    static double rthalf = {0.70710678118654752440};

    eps = DBL_EPSILON * 4.0;
    assert(approx(cosh(-1.0), (e + 1.0 / e) / 2.0));
    assert(approx(cosh(0.0), 1.0));
    assert(approx(cosh(1.0), (e + 1.0 / e) / 2.0));
    assert(approx(exp(-1.0), 1.0 / e));
    assert(approx(exp(0.0), 1.0));
    assert(approx(exp(ln2), 2.0));
    assert(approx(exp(1.0), e));
    assert(approx(exp(3.0), e * e * e));
    assert(log(1.0) == 0.0);
    assert(approx(log(e), 1.0));
    assert(approx(log(e * e * e), 3.0));
    assert(approx(log10(1.0), 0.0));
    assert(approx(log10(5.0), 1.0 - log10(2.0)));
    assert(approx(log10(1e5), 5.0));
    assert(approx(pow(-2.5, 2.0), 6.25));
    assert(approx(pow(-2.0, -3.0), -0.125));
    assert(pow(0.0, 6.0) == 0.0);
    assert(approx(pow(2.0, -0.5), rthalf));
    assert(approx(pow(3.0, 4.0), 81.0));
    assert(approx(sinh(-1.0), -(e - 1.0 / e) / 2.0));
    assert(approx(sinh(0.0), 0.0));
    assert(approx(sinh(1.0), (e - 1.0 / e) / 2.0));
    assert(approx(sqrt(0.0), 0.0));
    assert(approx(sqrt(0.5), rthalf));
    assert(approx(sqrt(1.0), 1.0));
    assert(approx(sqrt(2.0), 1.0 / rthalf));
    assert(approx(sqrt(144.0). 12.0));
    assert(approx(tanh(-1.0), -(e * e - 1.0) / (e * e + 1.0)));
    assert(approx(tanh(0.0), 0.0));
    assert(approx(tanh(1.0), (e * e - 1.0) / (e * e + 1.0)));
    puts("SUCCESS testing <math.h>, part 3");
    return (0);
    }
```

□

第8章 <setjmp.h>

8.1 背景知识

C程序设计语言不允许函数的嵌套定义，也就是说我们不能在一个函数内部定义另一个函数，就像下面的定义方式：

```
int f(void)
    {   /* outer function */
    int g(void)
        {   /* NOT PERMITTED */
        ...
```

这种限制让我们不能隐藏同一层上的函数名，所以在一个翻译单元内声明的所有函数彼此之间都是可见的。这并不是主要的缺陷，因为我们可以把属于不同翻译单元的函数进行分组并放置在不同的C源文件中，这样就可以限制它们之间的可见性了。

然而，因为这种设计，C语言在另一个方面遇到了麻烦。除了返回调用那个函数的表达式外，C语言没有提供其他将控制权转移到一个函数之外的比较简单的方法。对于大多数的函数调用来说，这样的限制很方便，因为我们需要用嵌套函数调用和返回的规则来帮助理解程序的控制流。不过，在某些情况下，这个规则的限制性太强了。有时，如果能一下子跳出一个或多个函数调用，程序会更易编写和理解。我们希望跳过一个正常的函数返回，把控制权交给更早的函数调用的某个地方。这通常是处理严重的错误的最好方式。

非本地的 *goto*

在Pascal中就可以这样做。一个嵌套的函数中可以包含一个*goto*语句，该语句可以把控制权转移给一个函数之外的标号。（C中的*void*类型的函数在Pascal中叫做过程，在这里也使用“函数”来表示Pascal中的过程。）这个标号可以在任何包含嵌套函数定义的函数中，如下所示：

```
function x: integer; {a Pascal goto example}
   label 99;
   function y(val: integer): integer;
       begin
       if val < 0 then
           goto 99;
       ...
```

在一个 Pascal 函数中，我们必须在声明所有嵌套的函数之前声明这些标号，然后翻译器才能识别出一个非本地的 *goto*。

同一个函数内的 *goto* 通常可以很容易地把控制权转移给具有相应标号的语句，而对于一个非本地的 *goto* 就有些麻烦了。它必须先终止当前活动的函数调用的执行，而这样做又涉及了释放所有的动态分配的空间和恢复前面的调用环境，Pascal 甚至要关闭与通过这种方式释放的所有 `file` 变量相关的文件。调用了包含该 *goto* 语句的函数的函数再次成为活动的函数。如果 *goto* 语句中命名的标号不在当前活动的函数中，这个过程就会重复。最终，正确的函数会再一次活动起来，并把控制权转移到具有相应标号的语句。调用包含 *goto* 语句的函数的表达式永远都不会完成它的执行。

Pascal 使用函数嵌套的某些规则来限制非本地 *goto* 语句。Pascal 语言不允许把控制权转移给不活动的函数，所以我们不能把控制权转移给一个未知的函数。后来证明这是 Pascal 语言比 C 语言做得好的一个方面。

标号变量

更老的 PL/I 语言使用了另外一种方法来解决这个问题，即允许声明标号变量。我们可以在一个上下文环境中将一个标号赋给这样一个变量，然后在另一个上下文环境中把该变量作为 *goto* 语句的目标使用。标号变量中存储了程序执行非本地 *goto* 语句需要的所有信息。（*goto* 不一定是非本地的，也可以把控制权转移给当前函数的当前调用中的标号。）

与 Pascal 语言所用的方法相比，PL/I 语言所使用的方法结构化方面很差。因为 PL/L 中所定义的标号变量可能没有初始化就在 *goto* 语句中使用了，或者赋给变量的标号可能指向一个已经终止的函数调用，不能用了。不论是哪种情况，造成的损失都很惨重。除非该实现可以在转移控制权之前确认标号变量的内容，否则，这种跳转就可能出错，而且这样的错误很难调试。

C 语言使用库函数来实现非本地控制转移。头文件 `<setjmp.h>` 提供了以下必需的机制。

`jmp_buf`
- 类型 `jmp_buf`，你可以把它当成标号数据对象类型看待。

`longjmp`
- 函数 `longjmp`，用来实现非本地控制转移。

`setjmp`
- 宏 `setjmp`，把当前调用的上下文信息存储到一个 `jmp_buf` 类型的数据对象中，并在你想把控制权传递给相应的 `longjmp` 调用的地方作标记。

在这点上，C 语言的机制甚至比 PL/I 语言的非结构化 *goto* 还要原始。我们所能做的工作就是记录下程序前期执行的某个位置，然后通过合适的 `jmp_buf` 数据对象为参数调用 `longjmp` 来返回到那个位置。但是如果这个数据对象没有初始化或者已经过期，那么就可能引发一场灾难。

longjmp 和 setjmp 都是很微妙的函数。它们强制干涉控制流和动态存储空间的管理。这两方面都是翻译器中非常复杂和难以编写的部分。这部分必须生成既正确又优化空间和运行速度的代码。优化经常涉及控制流的细微改变或者动态空间的使用。但是代码生成器又经常是在忽略了 longjmp 和 setjmp 的属性和行为的情况下工作的。

两个危险之处

C 标准指出了两个存在潜在危险的地方：

- 包含宏 setjmp 的表达式。
- 在执行 setjmp 的函数中声明的动态存储空间。

在这两种情况下，C 标准中的措辞有点让人不解，这是因为 C 标准试图限制危险的行为有强调这些危险。

其中一个危险之处在于表达式计算。一般电脑都有一定数量的寄存器，在对表达式进行求值的时候，寄存器用来保存中间结果。然而，在计算一个非常复杂的表达式时，这些寄存器可能不够用。这时用户就会迫使代码生成器把中间结果存储在动态存储空间中。

这样问题就来了。setjmp 必须要猜测多少“调用上下文”存储到 jmp_buf 数据对象中。某些寄存器的值需要保存起来。可以在函数调用过程中保存中间结果的寄存器是首选，因为 longjmp 调用可能出现在被调用的函数中。一旦程序要计算 setjmp 的值，它就需要这些中间结果来完成表达式求值。如果 setjmp 不能保存所有的中间结果，longjmp 调用的后续返回就会出现问题。

8

执行 setjmp

C 标准规定把包含 setjmp 的表达式作为子表达式使用。这种做法是为了排除某些可能把中间结果存储在 setjmp 未知（并且不可知）的动态存储空间中的表达式。因而你可以编写类似 switch (setjmp (buf))..., if (2<setjmp (buf))...,if (!setjmp (buf))... 的形式和表达式语句 setjmp (buf)。

你不可能写出比这些更复杂的形式了。注意，我们不能可靠地把 setjmp 的值赋给其他的变量，就像 n = setjmp (buf) 这样。这种表达式可能会正确地求值，但 C 标准并没有作这样的要求。

存储空间的回收

第二个危险的地方是关于处理执行 setjmp 的函数中的动态存储空间的问题。这样的存储空间出现在以下 3 种情况中。

- 为函数声明的参数。
- 所有使用 auto 存储类型说明符声明的数据对象，不管是显式的还是隐式的。
- 所有用 register 存储类型说明符声明的数据对象。

这些问题出现的原因是代码生成器可以选择把一些数据对象存储在寄存器中。这些寄存器组和存储表达式求值过程中的临时中间值的寄存器是无法区分的。因此，在一个 longjmp 调用中，setjmp 必须保存所有这些寄存器并且把它们恢复到一个早期的状态。这就意味着在 setjmp 的后续返回中，某些动态数据对象返回到了一个更早的状态。setjmp 的两次返回之间的存储值的任何变化都会丢失。

如果这样的行为是可预见的，那么它就是一种讨厌的异常情况。但问题是它是不可预见的。用户无法知道哪些参数和 auto 数据对象最后保存在寄存器中。甚至声明为 register 的数据对象也是不确定的。翻译器没有责任把任何这样的数据对象存储在寄存器中。因此，如果一个函数执行了 setjmp，并且一个 longjmp 调用把控制权转交回了函数，那么函数中声明的所有数据对象的值都是不确定的。这种事件处于一种无规律的状态。

易变的动态存储空间

X3J11 通过向语言中添加一个小功能来处理这个问题。我们可以把一个动态数据对象声明为 *volatile* 类型，然后翻译器就会更加小心地对待它们。这种类型的数据对象永远都不会存储到 longjmp 可以修改到的地方。当然，这种用法扩展了 *volatile* 的语义，但这确实有效。

8.2 C 标准的内容

<setjmp.h>

7.6 非本地跳转 <setjmp.h>

头文件 <setjmp.h> 定义了宏 setjmp，并且为了绕过正常的函数调用和返回规则声明了一个函数和一个类型[106]。

声明的类型是：

jmp_buf

jmp_buf

它是一个数组类型，适合存储恢复一个调用环境所需的信息。

并没有指定 setjmp 是一个宏还是一个用外部连接声明的标识符。如果为了访问一个实际的函数而取消了宏定义，或者程序使用名字 setjmp 定义一个外部标识符，则行为未定义。

7.6.1 保存调用环境

7.6.1.1 宏 setjmp

setjmp

概述

```
#include <setjmp.h>
int setjmp (jmp_buf env);
```

说明

宏 setjmp 将它的调用环境保存在它的 jmp_buf 类型的参数中，以供后面 longjmp 函数使用。

返回值

如果返回来自一个直接的调用，则宏 setjmp 返回零。如果返回来自对函数 longjmp 的调用，则宏 setjmp 返回一个非零值。

环境限制

宏 setjmp 的调用应该只出现在下面某个上下文环境中。

- 一个选择或者循环语句的整个控制表达式；

- 关系运算符或者等于运算符的其中一个操作数，另一个操作数是一个整值常量表达式，它的结果表达式是一个选择或者循环语句的整个控制表达式。
- 一元操作符！的操作数，它的结果表达式是一个选择或者循环语句的整个控制表达式。
- 是一个表达式语句（可能强制转换为 *void* 类型）的整个表达式。

7.6.2 恢复调用环境

longjmp

7.6.2.1 函数 longjmp

概述

```
#include <setjmp.h>
void longjmp(jmp_buf env, int val);
```

说明

函数 longjmp 使用相应的 jmp_buf 参数来恢复程序的相同调用中宏 setjmp 的最近一次调用保存的环境。如果没有这样的调用，或者包含宏 setjmp 调用的函数已经在其前终止执行[107]，则行为未定义。

所有可访问的对象从 longjmp 被调用的时候起就有确定值，但包含相应的宏 setjmp 调用的函数中的、没有 volatile 类型的和在 setjmp 与 longjmp 调用之间改动了的具有自动存储期的对象的值是不确定的。

因为它能绕过常规的函数调用和返回机制，所以函数 longjmp 可以在中断、信号和其他相关的函数的上下文环境中正确地执行。然而，如果在一个嵌套的信号处理程序（就是说，在一个信号处理的过程中又发生了另一个信号，这时调用的那个函数）内部调用 longjmp 函数，则行为未定义。

返回值

在 longjmp 完成之后，程序会继续执行，就好像相应的宏 setjmp 的调用返回了 val 指定的值。函数 longjmp 不能让宏 setjmp 返回零，如果 val 是零，则宏 setjmp 返回 1。

脚注

106. 这些函数对处理程序中的低级函数中遇到的异常情况很有用。
107. 例如，通过执行 return 语句，或者因为另一个 longjmp 调用使得执行转移到这一组嵌套调用中的更前的一个函数中的 setjmp 调用。

8.3 <setjmp.h> 的使用

不论什么时候，只要想绕过常规的函数调用和返回规则，就可以使用 <setjmp.h>。<setjmp.h> 提供的非本地跳转是一个很脆弱的机制，只有在非用不可的时候才使用它而且只能采用特殊的方式使用。我建议你使用以下这种标准方式：

- 把每个对 setjmp 的调用分离到一个独立的（小）函数中。那样，就会使出现动态声明的数据对象被 longjmp 调用恢复的情况减到最少。
- 在一个 *switch* 语句的控制表达式中调用 setjmp。
- 在 *switch* 语句的 *case* 0 中调用的函数（称作 process）中执行所有实际的过程。
- 通过执行 longjmp(1) 调用，在任意位置报告错误并且重新启动 process。
- 通过执行 longjmp(2) 调用，在任意位置报告错误并终止 process。

8

也可以添加其他的 *case* 标号来处理 longjmp 所允许的其他的参数值。一个最高级别的函数可能具有以下形式：

```
#include <setjmp.h>

static jmq_buf jmpbuf;

void top_level(void)
    {   /* the top-level function */
    for (; ; )
        switch (setjmp(jmpbuf))
            { /* switch on alternate returns */
        case 0: /* first time */
            process();
            return;
        case 1: /* restart */
             <report error>
             break;
        case 2: /* terminate */
             <report error>
             return;
        default:/* unknown longjmp argument */
             <report error>
             return;
             }
    }
```

这里我假设所有对 jmpbuf 的引用都在这个翻译单元内。否则，必须用一个外部连接来声明 jmpbuf。（丢掉存储分类关键字 static。）或者，必须向那些一定要访问 jmpbuf 的函数提供一个指向它的指针。

jmp_buf 参数

注意，jmp_buf 是一个数组类型。如果写了参数 jmpbuf，翻译器会把它改为指向数组的第一个元素的指针，因为那才是 setjmp 和 longjmp 所需要的。所以即使 jmpbuf 看起来传递的是值，实际上它是通过引用传递的。这就是 setjmp 在 jmpbuf 中存储调用环境的方式。

为了保持一致，应该把每一个形参都声明为 jmp_buf　buf，然后再定义相应的实参为 jmpbuf。不要把形参声明为 jmp_buf *pbuf 或者把实参写为 &jmpbuf。后者更清晰，但是它与调用 setjmp 和 longjmp 的传统习惯不一致。

如果选择 setjmp 的另一种使用方法，那么在一个尽可能小的函数中执行这个宏。如果翻译器不对 setjmp 进行特殊的处理，程序就有可能通过。但如果它发觉 setjmp 有问题，那么为了安全，这个宏的几乎所有代码都要进行优化。

如果在一个信号处理器的内部调用 longjmp，就会出现另外一个警告。第 9 章详细地讨论了这些问题。

8.4 <setjmp.h> 的实现

实现 setjmp 和 longjmp 的唯一可靠的方式就是用汇编语言来编写这些函数。你需要清楚地知道翻译器是怎样生成代码的，也需要执行一些不能用C语言安全表达的操作。

宏 _NSETJMP

图 8-1 显示了文件 setjmp.h。事实证明，对各种各样的标准 C 的实现，它已经足够了。它假设调用的上下文可以用一个整型数组来存储。即使当存储的上下文包含了各种类型的数据对象的时候，它通常也这样做。内部头文件 yvals 定义了宏 _NSETJMP，这个宏确定 jmpbuf 中的元素个数。

宏 _Setjmp

setjmp

注意 <setjmp.h> 是在另一个宏（或者函数）_Setjmp 的基础上来定义宏 setjmp，内部头文件 <yvals.h> 又一次提供了所需的信息。你可以把宏 _Setjmp 定义为用不同的名字调用一个现有函数的宏，或者你可以把 _Setjmp 声明为一个用汇编语言实现的函数。但你不能提供一个调用另一个函数的函数，（考虑一下这一点。）这也是我为定义宏 setjmp 提供很大灵活度的原因。你可以学习一下 PC 兼容机下的 Borland Turbo C++ 编译器，那是一个很好的例子。内部头文件 <yvals.h> 可能会包含下面的内容。

```
#define _NSETJMP    10
int _Setjmp(int *);
```

虽然前面有一些警告，但这里我还是给出了用 C 语言编写的 setjmp 和 longjmp，我这样做只是为了解释其中的原理，不要在一个严谨的实现中使用这些代码，因为它几乎不能工作，它只适用于那些具有特殊属性的实现。

- 调用函数的调用环境和其他动态分配的空间都存储在栈顶的一个连续的区域中。
- 调用环境包含所有 setjmp 保留和 longjmp 恢复的信息。可以通过复制固定数量的字符，来可靠地获得这些信息。

图 8-1 setjmp.h

```
/* setjmp.h standard header */
#ifndef _SETJMP
#define _SETJMP
#ifndef _YVALS
#include <yvale.h>
#endif
        /* macros */
#define setjmp(env) _Setjmp(env)
        /* type definitions */
typedef int jmp_buf[_NSETJMP];
        /* declarations */
void longjmp(jmp_buf, int);
#endif
```

8

- 部分调用环境是调用函数保存的帧指针，这些被保存的帧指针的位置跟一个声明的动态数据对象的位置之间有一个固定的偏移量，可以用来定位帧指针。
- 如果调用环境位于正确的位置，并且帧指针也被赋予了正确的值，那么函数就恢复为提供调用环境的调用者。

很多 C 语言的实现都满足这些假设，但并不是全部。这些函数恰好可以在 VAX 体系结构下工作。为了更好地解释它们是怎样工作的，我使用了一些参数来实现它们。对 VAX 来说，头文件 <yvals.h> 包含以下的宏定义：

```
#define _JBFP        1  /* int offset of frame pointer */
#define _JBMOV       60 /* number of bytes in calling context */
#define _JBOFF       4  /* byte offset of calling context */
#define _NSETJMP     17 /* number of ints in jmp_buf */
```

函数 setjmp

图 8-2 给出了文件 setjmp.c，它定义了 setjmp，但这个版本不是很完善。这个函数假设它可以把栈的一个连续的空间复制到 jmp_buf 数据对象中，并且能保存足够数量的调用环境。它声明了很多 register 数据对象，希望借此可以强制保存所有具有调用上下文的重要的寄存器。它制造了一个调用 dummy 函数的假象，这样就骗过了一些优化器，这些优化器可能断定那些寄存器从没有被使用过。

图 8-2 setjmp.c

```
/* setjmp function */
#include <setjmp.h>
#include <string.h>

static void dummy(int a, int b, int c, int d, int e,
    int f, int g, int h, int i, int j)
    {                                  /* threaten to use arguments */
    }

static int getfp(void)
    {                             /* return frame pointer of caller */
    int arg;

    return ((int)(&arg + _JBFP));
    }

int setjmp(jmp_buf env)
    {                             /* save environment for re-return */
    register int a = 0, b = 0, c = 0, d = 0, e = 0;
    register int f = 0, g = 0, h = 0, i = 0, j = 0;

    if (a)                             /* try to outsmart optimizer */
        dummy(a, b, c, d, e, f, g, h, i, j);
    env[1] = getfp();
    memcpy((char *)&env[2], (char *)env[1] + _JBOFF, _JBMOV);
    return (0);
    }
```
□

图 8-3
longjmp.c

```
/* longjmp function */
#include <setjmp.h>
#include <string.h>

static void dummy(int a, int b, int c, int d, int e,
    int f, int g, int h, int i, int j)
    {                                   /* threaten to use arguments */
    }

static void setfp(int fp)
    {                                   /* set frame pointer of caller */
    int arg;

    (&arg)[_JBFP] = fp;
    }

static int dojmp(jmp_buf env)
    {                                   /* do the actual dirty business */
    memcpy((char *)env[1] + _JBOFF, (char *)&env[2], _JBMOV);
    setfp(env[1]);
    return (env[0]);
    }

void longjmp(jmp_buf env, int val)
    {                                   /* re-return from setjmp */
    register int a = 0, b = 0, c = 0, d = 0, e = 0;
    register int f = 0, g = 0, h = 0, i = 0, j = 0;

    if (a)                              /* try to outsmart optimizer */
        dummy(a, b, c, d, e, f, g, h, i, j);
    env[0] = val ? val : 1;
    dojmp(env);
    }
```

函数 longjmp 图 8-3 是文件 longjmp.c。它定义了 longjmp 的一个更差的版本。这个函数把已保存的调用环境重新复制回栈中。它和 setjmp 一样分配寄存器并且又调用了另一个函数，目的是希望这种大量的复制不会和栈的任何有效使用的部分重叠。然后它设置帧指针，希望可以把控制权返回给调用 setjmp 的函数，而不是它的真正调用者。

如果一切顺利的话（很多原因都会造成一些问题），执行会在 setjmp 第一次调用的地方重新开始。这种情况下，setjmp 的返回值就是提供给 longjmp 的一个参数值。

这两个函数的完整的实现一定要更加严密。例如，它也会考虑到以下内容（关于其他的东西）：

- 浮点协处理器的状态；
- 信号处理器是否处于活动状态。

你会发现这些函数的正确版本使用了很多的技巧，只是变得更加可靠。

图 8-4 tsetjmp.c

```
/* test setjmp functions */
#include <assert.h>
#include <setjmp.h>
#include <stdio.h>

        /* static data */
static int ctr;
static jmp_buf b0;

static void jmpto(int n)
    {                                       /* jump on static buffer */
    longjmp(b0, n);
    }

static char *stackptr(void)
    {                                       /* test for stack creep */
    char ch;

    return (&ch;)
    }

static int tryit(void)
    {                                       /* exercise jumps */
    jmp_buf b1;
    char *sp = stackptr();

    ctr = 0;
    switch (setjmp(b0))
        {                                   /* jump among cases */
    case 0:
        assert(sp == stackptr());
        assert(ctr == 0);
        ++ctr;
        jmpto(0);                           /* should return 1 */
        break;
    case 1:
        assert(sp == stackptr());
        assert(ctr == 1);
        ++ctr;
        jmpto(2);
        break;
    case 2:
        assert(sp == stackptr());
        assert(ctr == 2);
        ++ctr;
        switch(setjmp(b1))
            {                               /* test nesting */
    case 0:
        assert(sp == stackptr());
        assert(ctr == 3);
        ++ctr;
        longjmp(b1, -7);
        break;
```

图 8-4 (续)

```
        case -7:
            assert(sp == stackptr());
            assert(ctr == 4);
            ++ctr;
            jmpto(3);
        case 5:
            return (13);
        default:
            return (0);
            }
    case 3:
        longjmp(b1, 5);
        break;
        }
    return (-1);
    }

int main()
    {                   /* test basic workings of setjmp functions */
    assert(tryit() == 13);
    printf("sizeof(jmp_buf) = %u\n", sizeof (jmp_buf));
    puts("SUCCESS testing <setjmp.h>");
    return (0);
    }
```

8.5 <setjmp.h> 的测试

图 8-4 是文件 tsetjmp.c，它似乎不只是一个功能测试，而更多的是对 setjmp 和 longjmp 的测试。我想你可能会自己试着用汇编语言实现这些函数。根据我的经历，要想找出这些代码中的 bug，必须要进行非常严密的测试，测试用例越繁琐越好。

栈蔓延

注意，这些代码反复地测试栈蔓延。如果不能把调用栈准确地恢复到一个早期的状态，就会出现这种情况。很多时候你会把各种垃圾留在栈中而且在很长一段时间内不去管它。只有当你的程序不经意间把栈空间用完，或者发生了其他形式的错误时，你才开始怀疑这样的问题。最好早点发现它们。

为了美观，程序也显示了类型为 jmp_buf 的数据对象的大小。当 testjmp.c 正确执行时，它会显示类似下面的信息：

```
sizeof (jmp-buf) = 20
SUCCESS testing <setjmp.h>
```

如果中间发生了错误，这个程序可能会暂停或者异常终止，它甚至会显示一条有用的错误信息。

8.6 参考文献

ISO/IEC Standard 71 85:1990 (Geneva:International Stantards Organization,1990). 这个标准定义了 Pascal 程序设计语言，这种语言允许使用非本地 *goto* 指向一个包含它的函数。

ISO/IEC Standard 6160:1979 (Geneva:International Stantards Organization,1979). 这个标准定义了 PL/I 程序设计语言，这种语言允许使用具有标号变量的非本地 *goto*。

8.7 习题

8.1 你使用的 C 翻译器是怎样定义 `jmp_buf` 类型的？你可以把它安全地用整型数组表示出来吗？如果可以，该数组必须有多少个元素？

8.2 编写可以在你的翻译器下工作的 `longjmp` 和 `setjmp` 版本。

8.3 修改在前面的习题中所编写的程序，检查一些比较明显的用法错误。

- 把一个校验和或者其他的签名存储在每一个 `jmp_buf` 类型的数据对象中，然后在你相信剩余的内容之前检查它们。
- 确保调用栈至少和它的内容存储在 `jmp_buf` 类型的数据对象中的时候一样大。

你还能想出其他什么样的检查方法？

8.4 [**难**] 异常处理程序是当报告或发生了一个异常时，可以获得控制权的代码段。在一个给定的上下文环境中，除了注册一个异常的编码值之外还要注册处理器。任何因为相同的异常编码值而被注册的异常处理程序都被屏蔽了（换句话说，在栈中注册了）。当上下文环境结束时，就可以把该处理程序注销，然后就会显示出所有前面的异常处理程序。一个处理程序可以注册处理任何情况的请求，也可以重新引发一个异常——把它向上传递给以前注册的处理程序。如果对一个给定的编码值，没有任何注册的处理程序，那么程序就会异常终止，输出一条讨厌的信息。设计函数 `when` 和 `raise` 来实现异常处理。`when` 可以注册和注销处理程序，`raise` 可以报告异常。使用这样的函数有什么好处？

8.5 [**难**] 实现你在上一道习题中设计的函数。

8.6 [**很难**] 定义可以消除本章前面描述的问题的 `setjmp` 和 `longjmp` 的语义。使得你可以从一个任意的表达式中调用 `setjmp`，而且所有的数据对象都不受 `longjmp` 调用的影响。相应地修改标准 C 翻译器。

第9章

<signal.h>

9.1 背景知识

信号是程序执行过程中发生的异常事件。同步信号的产生是因为程序自身的某些动作，例如除零或不正当地访问存储器。异步信号是由程序外部的行为引起的。比如有人敲击了提示键，或者另外一个程序（异步地执行）给你的程序发信号，都会引发一个异步信号。

程序不能屏蔽的信号要求立即得到处理。如果不对发生的信号指定一种处理方法，那么它就会被作为一种致命错误对待，然后程序就会以失败的状态终止执行。在某些实现中，这些状态反映了信号的类型。另外，C 标准库会在程序终止之前向标准错误流输出一条错误信息。

头文件 <signal.h>

头文件 <signal.h> 定义了一个无穷信号集的各种编码值。它也声明了两个函数：

raise

signal

- 函数 raise，报告一个同步信号；
- 函数 signal，可以指定一种信号的处理方法。

可以通过以下 3 种方式处理信号。

- 默认处理是终止程序，就像上面所描述的那样。
- 信号忽略是直接把它丢弃。
- 信号处理是把控制权转移给一个指定的函数。

信号处理程序

在最后一种情况下，那个指定的函数叫做信号处理程序。C 标准库在遇到一个报告的信号时，就会调用相应的信号处理程序，程序的正常执行就会被挂起。如果信号处理程序把控制权返回给调用者，程序就会从它被挂起的那个点继续执行。除了延迟和信号处理程序造成的任何变化，程序的行为不受影响。

这听起来像是一个很好的机制，但实际上并非如此。一个信号的发生又会在程序内部引入控制线程，这会引发程序的同步和可靠性操作方面的各种问题，但 C 标准对这两方面都不是很重视。C 程序从语言发展的早期就开始处理信号了。然而，一个可移植的程序却几乎不能安全地使用信号处理程序。

一个问题是C标准库本身。如果用一个有效的参数值调用（库函数），没有库函数会产生一个同步信号。但是库执行的时候却可能发生异步信号。例如，这个信号可能会通过一个打印操作挂起程序的执行。信号处理程序应该输出一条消息，但输出流可能会在一种迷惑的状态下结束。还没有一种方法可以在信号处理程序内部确定某一个库函数是否处于不安全的状态。

volatile 数据对象

另一个问题是关于声明为*volatile*类型的数据对象。这会警告翻译器，可能会有意想不到的实体访问这个数据对象，所以对这样的数据对象进行访问时一定要很小心。特别地，它也要知道把对volatile数据对象的访问移到某些*序列点*之外时不能进行优化。当然，一个信号处理程序就是一个意想不到的实体。因此，应该把在信号处理程序内访问的所有数据对象声明为*volatile*类型，如果信号是同步的而且发生在数据对象没有被访问的两个序列点之间，那么这样做是有好处的。然而，对于异步信号而言，多少保护都是不够的。信号不能仅仅被限制为只在某些序列点挂起程序的执行。

sig_atomic_t 类型

C标准对编写可靠的信号处理程序问题提供了部分的解决方法。头文件<signal.h>定义了sig_atomic_t类型，它是一个程序以原子方式访问的整数类型。一个信号在程序访问这种类型的数据对象的过程中，决不会挂起程序的执行。一个信号处理程序能和程序的其他部分共享的仅仅有声明为*volatile* sig_atomic_t类型的数据对象。

问题

作为一种传达信息的方法，信号还有很多有待改进的地方。标准C中对信号的详细说明是以它们在早期的UNIX操作系统下的行为为基础的。该系统在处理信号的方式上有很多严重的漏洞。

- 有多种信号可能会丢失。该系统没有把信号放到队列中，而只是记住了最后报告的信号。如果在信号处理程序处理一个信号之前，又有一个信号发生了，那么就会有一个信号被忽略。
- 一个程序甚至在它努力处理所有的信号的时候都有可能被终止。当控制权第一次传递给信号处理程序时，它对那个信号的处理恢复为默认行为，信号处理程序必须调用signal来使它自己复位为该信号的处理程序。当信号发生在进入信号处理程序和调用signal之间时，默认的处理程序就会得到控制权并终止程序。
- 目前还不存在一种机制可以专门终止一个信号的处理。在其他的操作系统中，程序输入一个特殊的状态，当信号处理程序报告完成当前的处理后，后续信号的处理才能顺利进行。在这样的系统中，其他的函数可能会适当地辅助对信号的正确处理。这些函数包括<stdlib.h>中声明的abort和exit以及<setjmp.h>中声明的longjmp。

另外，在任何计算机上，信号都是由各种奇怪的原因引起的。C标准中命名的这些原因是UNIX所支持的命名的一个子集，而这些又起源于为

PDP-11定义的中断和陷阱。把一个给定的计算机的信号源和标准C中定义的对应起来经常是任意的，把一个给定的操作系统的信号处理语义和标准C中的对应起来就更随意了。

C标准不得不削弱已经很弱小的UNIX信号的语义来使它可以适应各种各样的操作系统：

- 一个指定的信号可能永远都不会发生，除非用 raise 报告它；
- 一个指定的信号可能会被忽略，除非你用 signal 来激活它；

其他的就没什么了。

可移植性 因此，对 <signal.h> 中声明的函数不能完全安全地定义一种可移植的用法。原则上，可以为一个只有 raise 报告的信号指定一个处理程序，但很难想象它哪些方面比 <setjmp.h> 中声明的 setjmp 和 longjmp 工作得更好。另外，也不能保证一个指定的信号在C的所有实现中都不会被报告。所以，不管什么时候程序处理信号，它的可移植性都会受到限制。

9.2 C标准的内容

<signal.h>

7.7 信号处理 <signal.h>

头文件 <signal.h> 为处理各种各样的信号（程序执行过程中可能被报告的情况）声明了一个类型和两个函数，并定义了几个宏。

定义的类型是：

sig_atomic_t

```
sig_atomic_t
```

该类型是整数类型，声明为这种类型的对象可以作为一个原子实体被访问，即使有异步中断发生的时候也是如此。

定义的宏有：

SIG_DFL **SIG_ERR** **SIG_IGN**

```
SIG_DFL
SIG_ERR
SIG_IGN
```

它们展开为具有不同值的常量表达式，这些表达式的类型和函数 signal 的第二个参数及返回值的类型兼容，并且它们的值与所有声明的函数的地址都不相等。下面的每个宏都展开为正的整值常数表达式，它们是和各种指定的情况对应的信号编号。

SIGABRT SIGABRT 异常终止，比如执行了 abort 函数。

SIGFPE SIGFPE 错误的算术操作，比如除零或者导致溢出的操作。

SIGILL SIGILL 检测到无效的函数映象，比如一条非法指令。

SIGINT SIGINT 收到一个交互的提示信号。

SIGSEGV SIGSEGV 对存储器的无效访问。

SIGTERM SIGTERM 送到程序中的终止请求。

除了作为直接调用 raise 函数产生的结果，实现不需要生成这些信号中的任何一个。实现也可以使用宏定义来指定附加的信号和指向未声明的函数的指针，这些宏的名字分别以 SIG 和一个大写字母或 SIG_ 和一大写字母开头[108]。信号的完整集合、信号的语义和信号的默认处理方法是由实现定义的。所有的信号编号都应该是正数。

7.7.1 指定信号处理

signal

7.7.1.1 函数 **signal**

概述

```
#include <signal.h>
void (*signal(int sig, void (*func) (int) ) ) (int) ;
```

说明

函数 signal 通过 3 种方式来保证接收到的信号编号 sig 被依次处理。如果 func 的值是 SIG_DFL，那么就会使用默认的信号处理方式；如果 func 的值是 SIG_IGN，那么这个信号就会被忽略；否则，func 就指向一个函数，当这个信号发生时，就调用这个函数，这样的函数被称为信号处理程序。

当一个信号发生时，如果 func 指向一个函数，首先执行和 signal (sig, SIG_DFL); 等价的语句或者执行实现定义的信号块。（如果 sig 的值是 SIGILL，是否重置 SIG_DFL 是实现定义的。）然后执行和 (*func)(sig); 等价的语句。函数 func 可以通过 return 语句或者调用函数 abort、exit 或 longjmp 来终止。如果 func 执行 return 语句并且 sig 的值是 SIGFPE 或者任何其他的由实现定义的与计算异常对应的值，那么这种行为未定义；否则，程序会在它被中断的位置重新执行。

如果发生了一个信号，但不是调用函数 abort 或者 raise 产生的结果，而且信号处理程序调用了 signal 函数（函数的第一个参数是开启信号处理程序的信号编号）之外的其他标准库函数，或者引用了任何具有静态存储期的对象，而不是通过为一个 volatile sig_atomic_t 类型的静态存储变量赋值，那么这种行为未定义。而且，如果对 signal 函数的调用导致了返回值为 SIG_ERR，则 errno 的值是不确定的[109]。

在程序启动时，可以执行与下面等价的语句

```
signal(sig, SIG_IGN)
```

来处理某些按照实现定义的方式选择的信号。执行与下面等价的语句

```
signal(sig, SIG_DFL)
```

来处理实现定义的其他所有信号。

实现的行为不受调用函数 signal 的影响，即与没有库函数调用 signal 时的行为相同。

返回值

如果响应了请求的话，函数 signal 返回对指定的信号 sig 最后一次调用 signal 的 func 的值；否则，返回 SIG_ERR 的值并且在 errno 中存储一个正值。

参见： 函数 abort（7.10.4.1）和函数 exit（7.10.4.3）。

7.7.2 发送信号

raise

7.7.2.1 函数 **raise**

概述

```
#include <signal.h>
int raise(int sig);
```

说明

函数 raise 把信号 sig 送给正在执行的程序。

返回值

执行成功，函数 raise 返回零；否则，返回非零。

脚注

108. 参考“库的展望”（7.13.5）。信号编号的名字分别表示以下含义：异常中止、浮点异常、非法指令、中断、段越界和终止。
109. 如果所有的信号都是由异步信号处理程序产生的，那么这种行为未定义。

9.3 <signal.h> 的使用

从本质上说，信号处理是不可移植的。只有当必须指定一个已知的操作系统集的信号处理时，才能使用 <signal.h> 中声明的函数。不用费尽心思把代码变成通用的，这是不可能的。

信号处理

如果对一个信号的默认处理是可行的，那么就选择这种处理方法。使用自己的信号处理程序会降低可移植性并且可能导致程序对信号的误处理增多。如果必须提供某个信号的处理程序，对其进行分类如下：

- 不能返回值的信号处理程序，比如像 SIGFPE 这样报告算术异常或者像 SIGABRT 这样报告致命错误。
- 必须返回值的信号处理程序，像 SIGINT 这样，报告一个可能已经中断某种库操作的提示中断。

我们规定，第二类中包括不会立即导致程序终止的异步信号。几乎所有的信号都能归到这两类中。

不能返回值的信号处理程序最后会调用 abort、exit 或者 longjmp 来结束。当然，SIGABRT 的处理程序不能调用 abort 来结束。处理程序不应该调用 signal 使自己重建。如果程序没有终止，就把剩下的工作留给其他的函数。如果信号是异步的，要注意程序所有的输入输出，因为这样的操作也会中断库的执行。

必须返回的信号处理程序用一个 *return* 语句结束。如果它想让自己复位，那么应该在入口处立即返回。如果信号是异步的，就在一个 sig_atomic_t 类型的 *volatile* 数据对象中存储一个非零值。不要执行其他任何对正在执行的程序有副作用的动作，例如输入输出和访问其他的数据对象。

下面是一个异步信号处理程序的例子：

```
#include <signal.h>

static sig_atomic_t intflag = 0;

static void field_int(int sig)
    { /* handle SIGINT */
    signal(SIGINT, &field_int);
    intflag = 1;
    return;
    }
```

这个程序调用 signal (SIGINT,&field_int) 来创建处理程序。它会不时地执行下面的代码来检查是否发生了异步交互式提示中断：

```
if (intflag)
    { /* act on interrupt */
    intflag = 0;
    ...
    }
```

9

注意这些信号可能会在两个地方出错：

- 调用 signal 前，field_int 内部出现 SIGINT 可能会终止程序。
- 在测试和清除 intflag 之间出现的 SIGINT 可能会丢失。

这些都是信号固有的局限性。

下面是对信号的一个简单的描述，它适用于所有标准 C 的实现，但一个具体的实现可能会定义更多的东西。后面也显示了 <signal.h> 中定义的其他宏名的内容，这些宏都以 SIG 开头，它们都应该展开为表示其他信号的（小的）正整数。

SIGABRT

SIGABRT——当程序异常终止时，例如显式地调用 <stdlib.h> 中的 abort，就会产生这个信号。不要忽略这个信号。如果读者自己提供一个处理程序，就尽可能少地执行操作，用一个 *return* 语句或者调用 <stdlib.h> 中的函数 exit 来结束这个处理程序。

SIGFPE

SIGFPE——这个名字的原意是“浮点异常”，C 标准把这个信号推广为包括所有的算术异常，例如上溢、下溢或者除零。各种实现报告的异常类型区别也很大，但几乎没有实现会报告整型溢出。但也不能轻易地忽略这个信号，这个信号的处理程序不能有返回值。

SIGINT

SIGINT——这是报告异步交互式提示信号的常规方法。大多数系统都会提供一些组合键来生成这样的信号，例如 ctl-C、DEL 和 ATTN。它为提早结束烦人的循环提供了一种方便的途径。但是，异步信号可能会在一个程序的原子操作过程中打断它。如果处理程序不返回控制权，后来的程序可能会产生错误的行为。所以，忽略这个信号还是很安全的。

SIGSEGV

SIGSEGV——这个名字的原意是“段越界”，因为 PDP-11 把内存作为段的集合来管理。C 标准把这个信号推广为包括所有无效的存储访问引起的异常。例如程序尝试用错误的函数指示符或者左值访问 C 定义的所有函数和数据对象之外的存储区域，或者程序尝试把值存储在 *const* 类型的数据对象中。在这些事件中，程序不能继续安全地执行。不要忽略这个信号或者从它的处理程序中返回。

SIGTERM

SIGTERM——这个信号传统上是由操作系统或者另一个异步程序发出的，一定要把它当作终止程序执行客气而坚定的请求。它是一个异步信号，所以它可能会在程序的不恰当的位置点发生，所有有时候会想用上面描述的技术来使它延迟。忽略这个信号是很安全的，尽管这样做有点不太“礼貌”。

9.4 <signal.h> 的实现

图 9-1 所示是文件 signal.h，这里给出的头文件 <signal.h> 是最小的。例如 UNIX 系统就定义了几十个信号。而且在这点上，很多系统都努力向 UNIX 看齐。所以，即使它们不能产生这么多信号，它们还是定义了这些信号。尽管努力使行为保持一致，信号的选择和编码方式仍变化很大。这里我尽量选择使用最广泛的编码值。

头文件 <yvals.h>

和以前一样，我使用内部头文件 <yvals.h> 提供在不同的系统之间变化很大的参数。例如 SIGABRT 的编码和最有效的信号编码。这个实现中的某些函数使用宏 _NSIG 来确定无效的信号编码的最小正数。因此，头文件 <yvals.h> 在这里定义了两个重要的宏。对一个典型的 UNIX 系统来说，这两个定义是：

```
#define _SIGABRT  6
#define _SIGMAX   32
```

头文件 <signal.h> 对广泛使用的 UNIX 习惯做了一些让步。所以，它定义宏 SIG_ERR 和宏 SIG_IGN 的方式就有点不太舒服。在某些实现中，值 -1 和 1 可能是有效的函数地址。当然，这种可能性很小。假如这种可能性成立，连接器也一定可以避免这种情况，并且其他的值会更容易使用。（例如，signal 和 raise 的地址就不可能指定有用的信号处理程序。）但是这里选择的值是在 UNIX 实现中广泛使用的，其他的操作系统也模仿 UNIX 使用这些值。所以我选择它们是为了能和现有的体制兼容。

UNIX 版本

这种兼容性通常是有必要的。函数 signal 和 raise 几乎一定要修改以适应每个操作系统。UNIX 是一个极端的例子。在这种环境下，系统服务 signal 完成所有的工作。如果想访问同名的 C 可调用的函数，只要把这里给出的代码去掉就可以了。可以让其他的函数直接调用它。如果系统服务有一个私有的名字，例如 _Signal，那么，就可以把 signal 编写为：

```
/* signal function -- UNIX version */
#include <signal.h>

_Sigfun *_Signal(int, _Sigfun *)

-Sigfun *(signal)(int sig, _Sigfun *fun)
    { /* call the system service
    return (_Signal(sig, fun));
    }
```

当然也可以把它定义为 <signal.h> 中的一个屏蔽宏来使用。

函数 raise 会更难一点，它使用系统服务 kill 来给自己发送一个信号。（kill 早期只发送 SIGKILL 信号，所以不能只从它的字面上理解它的功能。）

9

图 9-1 signal.h

```
/* signal.h standard header */
#ifndef _SIGNAL
#define _SIGNAL
#ifndef _YVALS
#include <yvals.h>
#endif
        /* type definitions */
typedef int sig_atomic_t;
typedef void _Sigfun(int);
        /* signal codes */
#define SIGABRT_SIGABRT
#define SIGINT  2
#define SIGILL  4
#define SIGFPE  8
#define SIGSEGV 11
#define SIGTERM 15
#define _NSIG _SIGMAX                   /* one more than last code */
        /* signal return values */
#define SIG_DFL(_Sigfun *)0
#define SIG_ERR(_Sigfun *)-1
#define SIG_IGN(_Sigfun *)1
        /* declarations */
int raise(int);
_Sigfun *signal(int, _Sigfun *);
#endif
```

为了识别它自己，raise 还需要使用系统服务 getpid。假设这两个系统服务使用合适的私有名字，例如 Kill 和 _Getpid，那么就可以用下面的代码实现 raise：

```
/* raise function -- UNIX version */
#include <signal.h>

int _Getpid(void);
int _Raise(int, int);

int (raise)(int sig)
    { /* raise a signal */
    return (_Kill(_Getpid(), sig));
    }
```

这个也可以定义为 <signal.h> 中的一个屏蔽宏。

一般的版本 我选择的 signal 和 raise 的正式版本使用更广泛。但是它们没有提供标准 C 的信号和操作系统的信号之间的对应关系，因此就不能被推广。不过它们提供了添加具体操作系统编码的功能。那些不按照 UNIX 习惯处理信号的操作系统通常需要使用这些代码来区分 C 信号和操作系统信号之间的不同之处。

函数 raise 图 9-2 显示了文件 raise.c，它定义了一个不需要操作系统支持的 raise 版本。该函数包含了一个存储信号处理程序地址的数组 _Sigfun，这

个数组按照信号编码编排索引。开始时，数组的每一个元素都初始化为一个空指针，这恰好和 SIG_DFL 匹配，该值被 signal 用来表示默认处理。

raise 首先要确定信号编码是有效的。如果有效，函数就按照 _Sigtable 中对应的元素所指定的方式进行处理。默认的处理是向标准错误流输出一条单行信息并以不成功的状态终止程序。它对它所知道的信号进行命名而对于其他信号打印出编码值。如果想输出更多的错误提示信息，可以为附加的信号添加名字。

函数 signal

图 9-3 显示了文件 signal.c。它定义了函数 signal，该函数与上面的函数 raise 结合使用。它所做的就是确认它的参数有效并且用一个有效的函数指针来代替 _Sigtable 中的适当的项。（如果指针不和 SIG_ERR 匹配，就认为它是有效的，但这种检测非常不牢靠。）

声明 _Sigtable

注意这个文件中对 _Sigtable 的声明。我通常的做法是把这样的声明放在一个头文件中，所有要使用它的 C 源文件都要包含这个头文件。在这种情况下这个头文件可以是 <signal.h>，但是只有当某个屏蔽宏引用了它时才可以是 <signal.h>。所以更可行的是，这个头文件为某个具有 "xsignal.h" 这样名字的内部头文件。然而，我不能仅仅为了一个声明而创建一个头文件，任何风格在实际使用中都有例外。

硬件信号

产生“硬件信号”时，某些系统指定的代码需要得到控制权，可以把这些代码添加到 signal 中。硬件信号是操作系统或者计算机本身报告的信号，所以这里一定要小心。很多系统都会把控制权转移到指定的地址，但不会遵守 C 函数调用和返回的规则。所以通过这种方式处理的每一个信号都必须使用一些汇编代码。

告诉操作系统（或者计算机）把控制权转移给汇编语言信号处理程序，让处理程序保存所有有用的上下文环境，并且调用通过适当的协议调用指定的 C 函数。它可以从一个静态数据对象中确定一个地址，当然，你知道如何用 C 和汇编语言来访问这个数据对象。如果 C 函数返回了，汇编语言信号处理程序就会反过来把控制权转移给中断程序。

一些操作系统要求要求报告信号处理程序何时完成。对于一个可以返回的信号处理程序，这相对比较简单，汇编语言信号处理程序可以在退出的时候做任何需要做的事情。但是不要忘了，信号处理程序也可以通过调用 <stdlib.h> 中的 abort 或者 exit 或者通过调用 <setjmp.h> 中的 longjmp 来终止执行。为了可以正确处理所有的情况，我们还要适当地做一些工作。

9

图 9-2 raise.c

```
/* raise function -- simple version */
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>

        /* static data */
_Sigfun *_Sigtable[_NSIG] = {0} ;                    /* handler table */

int (raise)(int sig)
    {                                               /* raise a signal */
    _Sigfun *s;

    if (sig <= 0 || _NSIG <= sig)
        return (-1);                                    /* bad signal */
    if ((s = _Sigtable[sig]) != SIG_IGN && s != SIG_DFL)
        {                                  /* revert and call handler */
        _Sigtable[sig] = SIG_DFL;
        (*s)(sig);
        }
    else if (s == SIG_DFL)
        {                                         /* default handling */
        char ac[10], *p;

        switch (sig)
            {                         /* print known signals by name */
        case SIGABRT:
            p = "abort";
            break;
        case SIGFPE:
            p = "arithmetic error";
            break;
        case SIGILL:
            p = "invalid executable code";
            break;
        case SIGINT:
            p = "interruption";
            break;
        case SIGSEGV:
            p = "invalid storage access";
            break;
        case SIGTERM:
            p = "termination request";
            break;
        default:
            *(p = &ac[(sizeof ac) - 1]) = '\0';
            do *--p = sig % 10 + '0';
                while ((sig /= 10) != 0);
            fputs("signal #", stderr);
            }
        fputs(p, stderr);
        fputs(" -- terminating\n", stderr);
        exit(EXIT_FAILURE);
        }
    return (0);
    }
```

□

图 9-3
signal.c

```
/* signal function -- simple version */
#include <signal.h>

        /* external declarations */
extern _Sigfun *_Sigtable[_NSIG];

_Sigfun *(signal)(int sig, _Sigfun *fun)
    {                            /* specify handling for a signal */
    _Sigfun *s;

    if (sig <= 0 || _NSIG <= sig || fun == SIG_ERR)
        return (SIG_ERR);                        /* bad signal */
    /* add machine-dependent handling here */
    s = _Sigtable[sig], _Sigtable[sig] = fun;
    return (s);
    }
```

9.5 <signal.h> 的测试

图 9-4 显示了文件 tsignal.c，它做的工作很少，因为信号的可移植的属性太少了。该文件所做的全部工作就是用 SIGFPE 测试 signal 和 raise 基本的工作情况。这些代码假设程序执行的时候没有其他的程序报告这个信号。这个假设是非常安全的，但是 C 标准并没有这样的保证。这个测试程序也可以检测是不是定义了各种宏，包括 sig_atomic_t 类型。但是，它并不去检查相关的语义。

因此，程序以字节为单位显示了 sig_atomic_t 的大小。如果一切进行顺利的话，程序会显示以下内容：

```
sizeof (sig_atomic_t) = 2
SUCCESS testing <signal.h>
```

9.6 参考文献

PDP-11/70 Processor Handbook(Maynard, Mass.: Digital Equipment Corporation, 1976). PDP-11 的陷阱和中断是 UNIX 最早定义信号的灵感之源。它可以让我们更好地理解 UNIX 信号的名字和语义。

9.7 习题

9.1 列出你使用的 C 翻译器定义的信号编码。能用一句话描述每个信号表示的含义吗？

9.2 对于你使用的翻译器定义的信号编码，设计可以引发每个信号的测试程序。

9.3 在哪种情况下你会关心是否存在没有报告的信号？

图 9-4
tsignal.c

```
/* test signal functions */
#include <assert.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>

        /* static data */
static int sigs[] = {
    SIGABRT, SIGFPE, SIGILL, SIGINT, SIGSEGV, SIGTERM};
static void (*rets[])(int) = {SIG_DFL, SIG_ERR, SIG_IGN};
static sig_atomic_t atomic;

static void field_fpe(int sig)
    {                                               /* handle SIGFPE */
    assert(sig == SIGFPE);
    puts("SUCCESS testing <signal.h>");
    exit(EXIT_SUCCESS);
    }

int main()
    {                   /* test basic workings of signal functions */
    printf("sizeof (sig_atomic_t) = %u\n",
        sizeof (sig_atomic_t));
    assert(signal(SIGFPE, &field_fpe) == SIG_DFL);
    assert(signal(SIGFPE, &field_fpe) == &field_fpe);
    raise(SIGFPE);
    puts("FAILURE testing <signal.h>");
    return (EXIT_FAILURE);
    }
```

9.4 修改 signal 和 raise，使它们能适应你的 C 翻译器，尽可能多地处理硬件信号。

9.5 编写信号 SIGABRT 的处理程序，使它可以显示*逆踪迹*（trace back）——活动函数的列表，这个列表按照它们的调用顺序逆序显示。这种功能有什么用途？

9.6 **[难]** 找出 C 标准库中不应该被一个信号中断的关键区域。如果这些区域活动的时候有信号被报告，就推迟对该信号的处理，直到关键区域的活动结束。这种功能有什么用途？

9.7 **[很难]** 实现信号的新的语义，要保证以下几点：

- 信号不能被复制或者丢失；
- 信号按照报告的顺序进行处理；
- 一个程序在某个点之后一定可以处理所有报告的信号；
- 关键区域可以被保护不被中断打扰；
- 一个信号处理程序可以安全地跟程序的其他部分通信。

第 10 章

<stdarg.h>

10

10.1 背景知识

C 语言有一个强大的功能，就是它允许定义可接受一个可变参数列表的函数。当然，其他语言也有这样的函数，但是这种函数的数目是固定的，而且所有的都是内嵌到语言中的特殊函数，不能自己定义额外的函数。

为了访问一个可变参数表中额外的参数，就需要使用 <stdarg.h> 中定义的宏。它们允许在任何时候从头到尾地遍历一个附加参数列表。在遇到每一个参数之前，必须知道它的类型。但是在一个给定的调用发生之前，不必知道它的细节。可以使用一个固定的参数，例如一个格式串，来确定参数的数目和类型。

头文件 <stdarg.h> 是 X3J11 委员会的发明，它很大部分都以头文件 <varargs.h> 为基础。这个头文件是 Andy Koenig 为了增强 UNIX 操作系统的可移植性而开发的。当时曾几次尝试摆脱实现对遍历可变参数表的依赖，<varargs.h> 就是其中的一个版本，也是众所周知的一个。其中的思想是通过把差别隐藏在宏里来构造一个可移植性更强的公共操作。

历史 在很早的时候，这样的隐藏是不必要的，因为当时 C 是 PDP-11 专用的语言。每个人都知道 Dennis Ritchie 的编译器是如何安排内存中的参数列表的。在指针算术中，参数的转移非常简单。指针的大小和 *int* 类型的大小相同，并且结构不允许作为参数，这一点也很有用。那就是说可以把参数当作 *int*、*long* 或者 *double* 类型来对待。因为在 PDP-11 中，*double* 的存储排列和 *int* 的相同，所以为了保证适当的存储对齐不必担心参数列表中留出很多空隙。

结构参数和各种大小的指针的出现带来了混乱。即使不想编写可移植的代码，我们仍然希望代码具有很强的可读性，这就强烈地需要使用一个符号来隐藏遍历可变参数表的杂乱的细节。

随后出现了一些C的实现，它们的设计目标是让C语言可以和像FORTRAN这样的老的程序设计语言一起工作。对这样的实现来说，有时候就需要一个调用序列，这个调用序列和PDP-11的区别很大。在内存中，参数列表有时候向下延伸，而不是向上，还有某些参数列表涉及了指向实际参数值的中间指针。这时，隐藏访问参数的细节从便利性提升到了必要性。

头文件 <stdarg.h>

X3J11委员会感到有责任在某些小方面修改存在的宏，这也是C标准用一个新的名字命名一个标准头文件的原因。<stdarg.h> 和 <varargs.h> 的差别很大，足以给那些使用旧的头文件的程序（和程序员）造成困惑。该委员会经过很多讨论，就是为了让 <stdarg.h> 的功能更像是这个语言的一部分。然而，最后委员会通过投票把遍历可变参数表的机制作为宏保留了下来。

X3J11实际上做的是尽可能推广这些宏。他们的做法是以某种方式定义这些宏，使得所有已知的C的实现无需大的修改就可以跟它们保持一致。仍有某些实现不得不修改它们的翻译器来提供关键的信息或者操作。然而，大部分实现都不需要翻译器的帮助就可以支持 <stdarg.h>。

限制

<stdarg.h> 中定义的宏被强加了一些限制。这些限制表面上看起来不是很严重，但对某些实现来说，确实很严重。不过，每一个限制的引入都是为了满足至少一个严格的C实现的需要，例如：

宏 va_start

- 一个函数必须至少声明一个固定的参数。宏 va_start 引用了最后一个固定参数所以它能够对可变参数表进行定位。

宏 va_arg

- 不能在 va_arg 中没有函数原型的情况下指定会“增宽”（widen）的参数类型。例如，必须用 doulbe 来代替 float。这些宏不能复制适用于可变参数表的改变参数类型的规则。
- 在 va_arg 中，只能使用某些参数类型。这是因为很多宏的实现需要生成一个相关的指针类型，这个指针类型只是在原类型后面加上一个 *。C语言中的类型编写规则是有名的自闭型，所以对这样简单的工作方式实现起来也非常麻烦。

宏 va_end

- 一个函数在返回到它的调用者之前一定要调用 va_end。这是因为某些实现在函数返回之前需要整理控制信息。

但是，总而言之，<stdarg.h> 中定义的宏能够很好地工作，并且它们提供了现代编程语言中仅有的一个非常强大的功能。

10.2 C标准的内容

<stdarg.h>

7.8 可变参数 <stdarg.h>

头文件 <stdarg.h> 声明了一种类型且定义了 3 个宏，这样就可以提前访问一个参数表，调用函数在被编译时并不知道这个参数表中参数的数目和类型。

调用的函数具有可变数目和可变类型的参数。就像 6.7.1 中描述的那样，它的参数表包含一个或者多个参数。在这种访问机制中，最右边的参数起着特殊的作用，它由这个描述中的 *parmN* 指定。

声明的类型是

va_list

```
va_list
```

它是一个适合保存宏 va_start、va_arg 和 va_end 所需要的信息的类型。如果要访问不同的参数，那么调用函数要声明一个 va_list 类型的数据对象（假设该数据对象名为 ap）。对象 ap 可能做为参数传递给另一个函数。如果那个函数对参数 ap 调用了宏 va_arg，那么 ap 在调用函数中的值是不确定的，而且在其他对 ap 的引用之前会把它传递给宏 va_end。

7.8.1 可变参数表访问宏

本条中描述的宏 va_start 和 va_arg 将被作为宏实现，而不是实际的函数。并没有指定 va_end 是一个宏还是一个用外部连接声明的标识符。如果掩盖一个宏定义来访问实际的函数或者程序定义了一个名为 va_end 的外部标识符，那么这种行为未定义。如果要访问可变参数，那么宏 va_start 和 va_end 会在接受可变数目的参数的函数中被调用。

va_start

7.8.1.1 宏 va_start

概述

```
#include <stdarg.h>
void va-start (va-list ap, parmN);
```

说明

要在访问所有未命名的参数之前调用宏 va_start。

宏 va_start 对 ap 进行初始化，以便后面 va_arg 和 va_end 对它的使用。

参数 *parmN* 是函数定义（在……之前的那个）的可变参数表中最右边参数的标识符。如果用 register 存储类别，一个函数或者数组类型，或者和应用默认的参数提升产生的类型不兼容的类型来声明参数 *parmN*，那么这种行为未定义。

返回值

宏 va_start 没有返回值。

va_arg

7.8.1.2 宏 va_arg

概述

```
#include <stdarg.h>
type va-arg(va-list ap, type);
```

说明

宏 va_arg 展开为一个表达式，这个表达式的类型和值跟调用的下一个参数的相同。参数 ap 应该和 va_start 初始化的 va_list ap 相同。va_arg 的每一次调用都会修改 ap，这样后继参数的值就会依次返回。参数类型是指定的类型名字，使得指向指定类型的数据对象的指针的类型可以简单地通过在类型后面加一个后缀 * 来获得。如果实际不存在下一个参数，或者类型和实际存在的下一个参数（根据默认的参数提升规则来提升）的类型不兼容，那么这种行为未定义。

返回值

第一次调用 va_start 之后，对 va_arg 的第一次调用返回的是 *parmN* 指定的参数后面的参数值。后面的调用依次返回剩下的参数值。

10

va_end

7.8.1.3 宏 **va_end**

概述

```
#include <stdarg.h>
void va_end(va_list ap);
```

说明

宏 va_end 促进函数的正常返回，该函数的可变参数表被初始化了 va_list ap 的 va_start 的扩展所引用。宏 va_end 可能会对 ap 进行修改，这样它就不再有用（中间不能再一次执行 va_start）。如果没有执行相应的宏 va_start，或者在返回之前没有调用 va_end，那么这种行为未定义。

返回值

宏 va_end 没有返回值。

例子

函数 f1 把一个参数表（参数个数不大于 MAXARGS）放到一个数组中，其中这些参数是指向串的指针。然后把数组作为一个参数传递给函数 f2。指针的个数由函数 f1 的第一个参数指定。

```
#include <stdarg.h>
#define MAXARGS  31

void f1(int n_ptrs, ...)
{
      va_list ap;
      char *array[MAXARGS];
      int ptr_no = 0;

      if (n_Ptrs> MAXARGS)
            n_Ptrs = MAXARGS;
      va_start(ap, n_Ptrs);
      while (ptr_no < n_ptrs)
           array[ptr_no++] = va_arg(ap, char *);
      va_end(ap);
      f2(n-ptrs, array);
}
```

对 f1 的每一次调用都要有可见的函数定义或者类似下面的声明：

```
void f1 (int, ...);
```

10.3 <stdarg.h> 的使用

使用 <stdarg.h> 中声明的宏遍历一个可变参数表。这些宏一定要能满足各种实现的需求，因此，它们有以下的限制。

- 必须明确地声明一个函数具有一个可变参数表。（把它叫做 f。）这就意味着它的参数表一定以省略号（, . . .）结尾，在它的定义和声明中都是如此。而且，对这个函数的所有调用都应该在用这种方式声明函数的函数原型的范围内。
- 声明函数时必须至少有一个固定的参数，最后一个固定参数引用时习惯上称为 parmN。
- 必须声明一个 va_list 类型的数据对象，习惯上称为 ap。当然，这个数据对象在函数内一定是可见的。
- 必须在 f 内执行 va_start(ap, parmN)，在此之后才能执行 va_list 或者 va_end。

- 然后可以在函数中或者它调用的任意函数中执行 va_arg(ap,*T*)。当然，必须为每一个参数按照它们在函数调用中出现的顺序指定适当的类型。注意，va_arg 是一个右值宏。不能把这个宏调用作为一个左值来修改存储在参数数据对象中的值。
- 不能编写类型 *T*，使得当它作为一个参数传递时范围会变大。用 *double* 来代替 *float*，用 *int* 或者 *unsigned int* 来代替 *char*、*signed char*、*unsigned char*、*short* 和 *unsigned short*。对于和 *int* 的大小相同的 *unsigned short*，用 *unsigned int* 代替；对于表示为非负值，和 *int* 大小相同的字符类型，用 *unsigned int* 代替。
- 可以编写类型 *T*，它仅仅通过在后面加一个 * 就可以转换为指针类型。例如，类型说明符 int 和 char* 是有效的，类型说明符 char(*)[5] 是无效的。作为一般的规则，要注意那些包含圆括号或者方括号的类型说明符。
- 如果前面执行了 va_start，那么必须在 f 内执行 va_end。一旦执行了 va_end，就不能再执行 va_arg 了，除非首先执行 va_start 以启动一次重扫描。在这种情况下，必须在函数返回前再执行一次 va_end。

如果所有的这些听起来太消极了，那么可以考虑一个积极的例子。这里有一个对 <stdio.h> 中声明的函数 fputs 推广后的函数。这个函数把一个以空字符结束的字符串写到指定的输出流中，如：

```
fputs("this is a test", stdout);
```

而这个叫做 va_fputs 的函数，可以把任意数目的串写到指定的流中，如：

```
va_fpute(stdout, "this is", "a test", NULL);
```

在这个例子中，这两个函数向 stdout 流写入相同的输出。

可以这样编写 va_fputs:

```
#include <stdarg.h>
#include <stdout.h>

int va_fputs(FILE *str, ...)
    {  /* write zero or more strings */
    char *s;
    int status = 0;
    va_list ap;

    va_start(ap, str);
    while ((s = va_arg(ap, char *)) != NULL)
        if (fputs(s, str) < 0)
            status = EOF;
    va_end(ap);
    return (status);
    }
```

10

可以按照这种样式来处理一系列的可变参数表，甚至可以在一个独立的函数中处理可变参数表。当然，在调用函数之前要保证执行了 va_start。然后，在函数返回的时候执行 va_end。

重扫描

如果想重新扫描可变参数表，那么必须要更加小心。当然，首先要执行 va_start 来启动每一次重扫描。在函数返回之前，并且至少执行了一次 va_start 之后执行 va_end。推荐一种更加安全的方式——在同一个循环内执行 va_start 和 va_end。这样，就更能保证在应该执行 va_end 的时候执行它。

很多实现不需要 va_end，这个宏就展开什么都不做的代码。这就意味着使用这个宏的过程中出现的所有错误都会变成一个定时炸弹，不知道什么时候会出错。而且每过一年，寻找和改正它们的花费就变得更大。所以，最好预先尽力消除这些错误。

va_list 参数

另一个危险隐藏在调用一个具有 ap（va_list 类型的数据对象）参数的函数中。在某些实现中，它可能是数组类型，这就说明函数参数实际上变成了一个指向 va_list 数组首元素的指针。当被调用的函数执行 va_arg 时，这个数据对象在调用函数（上面称为 f）中改变了。在其他的实现中，va_list 不是数组类型，也就是说参数 ap 就像它外表那样通过值传递。当被调用的函数执行 va_arg 时，调用函数 f 中的这个数据对象不会改变。

如果处理了被调用函数中的所有参数，这些区别就不重要了。然而，如果在具有“相同”ap 的不同的函数调用中执行 va_arg，情况就不同了。实际上，如果代码要求 va_list 数据对象共享或者不共享，那么就会出现问题。

可以保证以下需要的行为：

- 如果 va_list 数据对象一定要共享，那么把参数写为 &ap，把相应的参数声明名 va_list *pap。在函数内，执行 va_arg(*pap,*T*) 来访问可变参数表中的每一个参数。
- 如果 va_list 数据对象不能被共享，那么把参数写为 ap，把相应的参数声明为 va_list xap。在函数内，声明一个 va_list ap 的数据对象并执行 memcpy(ap,xap, sizeof(va_list))。(memcpy 在 <string.h> 中声明)。执行 va_arg(ap, *T*) 来访问可变参数表中的每一个参数。

无论 va_list 定义为什么类型，这两种方法都可行。

10.4 <stdarg.h> 的实现

图 10-1 显示了文件 stdarg.h，它是实现 <stdarg.h> 所需的唯一代码。这里假定它可以在一个具体的标准 C 实现下工作。

假设 这种方法假设如下：

- 一个可变参数表在内存中占据了一个连续的字符数组。
- 后继的参数占据着字符数组更高位的后继元素。
- 一个参数占据的空间开始于 2^N 字节的整数倍的存储边界。
- 存储空间的大小是可以表示这个参数的 2^N 字节的最小倍数。
- 存储空间中留下的任何“空隙”总是在参数数据对象的开头或者结尾。

这些假设在很多标准 C 实现上都成立。

头文件 <yvals.h> 和往常一样，内部头文件 <yvals.h> 定义了描述不同系统之间的变化的宏。对头文件 <stdarg.h> 来说，有两个相关的参数：

宏 _AUPBND

- _AUPBND 是一个在可变参数表内部确定存储边界的屏蔽宏，它的值为 2^N-1。

宏 _ADNBND

- _ADNBND 是一个确定存储空隙是否在一个参数数据对象的开端或者结尾的屏蔽宏。如果空隙在尾处，它的值为 2^N-1；否则为 0。

Borland Turbo C++ 编译器就是一个简单的例子。对于该实现，头文件 <yvals.h> 包含了下面的定义：

```
#define -AUPBND 1
#define -ADNBND 1
```

图 10-1 stdarg.h

```
/* stdarg.h standard header */
#ifndef _STDARG
#define _STDARG
#ifndef _YVALS
#include <yvals.h>
#endif
        /* type definitions */
typedef char *va_list;
        /* macros */
#define va_arg(ap, T) \
    (*(T *)(((ap) += _Bnd(T, _AUPBND)) - _Bnd(T, _ADNBND)))
#define va_end(ap)    (void)0
#define va_start(ap, A)\
    (void)((ap) = (char *)&(A) + _Bnd(A, _AUPBND))
#define _Bnd(X, bnd) (sizeof (X) + (bnd) & ~ (bnd))
#endif
```

□

10

我发现在 Sun UNIX 工作站下的 GNU C 编译器中，有必要在一个参数之前指定一个存储空隙。对这个系统而言，_AUPBND 的值为 3，但是 _ADNBND 的值为 0。

va_list 类型

现在我们就能理解 stdarg.h 中涉及的技巧了。va_list 类型只是一个指向 *char* 的指针，这样的数据对象保存一个指向下一个参数空间的起始位置的指针。

va_start
_Bnd

宏 va_start 跳过已命名的参数，也就是最后一个固定参数。它使用内部宏 _Bnd 来把它的参数大小近似为 2^N 字节的一个倍数。

宏 va_arg

宏 va_arg 是这些宏中技巧性最强的一个。该宏首先通过增加 va_list 数据对象的内容来使它指向下一个参数空间的起始位置，然后再退回来指向当前参数的起始位置，然后通过强制类型转换把这个指针值转换为指定类型的指针，最后解引用这个指针以访问存储在数据对象中的值。(在这个实现中，va_arg 是一个左值，不要指望其他的实现中也是这样。)

宏 va_end

宏 va_end 在这个实现中什么都不做，它展开为一个占位符表达式 (void) 0。

10.5 <stdarg.h> 的测试

图 10-2 显示了文件 tstdarg.c，它测试 <stdarg.h> 中定义的宏。函数 tryit 接受一个具有各种参数类型的可变参数表。一个格式串参数会告诉这个函数想要的东西，和 <stdio.h> 中声明的打印和扫描函数差不多。

我发现不止一个实现不能正确地处理 Cstruct 类型的数据对象。它是包含一个单一字符的结构。并不是每个人都能记住一个参数会那样小。

因此，这个程序显示了 va_list 类型的数据对象的大小（以字节为单位)。如果一切顺利的话，测试程序会输出类似于下面的东西：

```
sizeof (va_list) = 4
SUCCESS testing <stdarg.h>
```

10.6 参考文献

UNIX Programmer's Reference Manual,4.3 Berkeley Software Distribution VirtualVAX-11 Version (Berkeley, Ca.: University of California, 1986). 这里讨论的是头文件 <stdarg.h> 的原型 <varargs.h> 的起源。

图 10-2 tstdarg.c

```
/* test stdarg macros */
#include <assert.h>
#include <stdarg.h>
#include <stdio.h>

        /* type definitions */
typedef struct {
    char c;
    } Cstruct;

static int tryit(const char *fmt, ...)
    {                                /* test variable argument list */
    int ctr = 0;
    va_list ap;

    va_start(ap, fmt);
    for (; *fmt; ++fmt)
        switch (*fmt)
            {                            /* switch on argument type */
        case 'i':
            assert(va_arg(ap, int) == ++ctr);
            break;
        case 'd':
            assert(va_arg(ap, double) == ++ctr);
            break;
        case 'p':
            assert(va_arg(ap, char *)[0] == ++ctr);
            break;
        case 's':
            assert(va_arg(ap, Cstruct).c == ++ctr);
            }
    va_end(ap);
    return (ctr);
    }

int main()
    {                        /* test basic workings of stdarg macros */
    Cstruct x = {3};

    assert(tryit("iisdi", '\1', 2, x, 4.0, 5) == 5);
    assert(tryit("") == 0);
    assert(tryit("pdp", "\1", 2.0, "\3") == 3);
    printf("sizeof (va_list) = %u\n", sizeof (va_list));
    puts("SUCCESS testing <stdarg.h>");
    return (0);
    }
```
□

10

10.7 习题

10.1 通过阅读文档确定你的 C 翻译器是怎样在一个可变参数表中存储参数的？文档告诉你的知识足够多吗？

10.2 通过阅读你的 C 翻译器提供的头文件 <stdarg.h>，确定该翻译器是怎样在一个可变参数表中存储参数的？这个文件告诉你的足够吗？

10.3 通过检查测试程序 tstdarg.c（图 10-2）生成的代码来确定你的 C 翻译器是怎样在一个可变参数表中存储参数的？那些知识足够吗？如果不够，向程序中添加点东西来提供这些缺失的信息。

10.4 修改本章中出现的代码，使头文件 <stdarg.h> 可以在你使用的 C 翻译器下工作。

10.5 编写函数 char *scat(char *dest, const char *src, . . .)。这个函数可以把一个或者多个字符串拼接起来并把它们写入 dest 中。第一个字符串的起始位置是 src，一个空指针结束这个列表。函数返回一个指针，指向那个起始位置在 dest 的字符串中表示结束的空字符。

10.6 **[难]** 假设你想测试一个参数是否在一个可变参数表中。如果在，你想知道它的类型。描述一个可以让你实现这种功能的符号。

10.7 **[很难]** 实现你在上题中描述的符号。

第 11 章 <stddef.h>

11.1 背景知识

头文件 <stddef.h> 是 X3J11 委员会在制定 C 标准过程中的又一发明。这个名字遵循了 C 标准库对头文件命名的隐晦模式。它暗含的意思就是“标准定义”。

这个头文件中的定义也可以放在另一个头文件 <stdlib.h>。后者也是该委员会的发明。它也用自己含糊的名字说明了它是声明各种函数的地方，不管是旧的还是新的，只要和其他标准头文件没有直接联系，都可以放在这个头文件中。创造这样两个包罗万象的存储室可能看上去很愚蠢，不过，委员会有它的理由。

独立与宿主

X3J11 委员会的某些成员坚持认为，即使在一个*独立环境*中，C 语言也应该有它的用途。独立的环境是一个由于种种原因而不能支持完整的 C 标准库的环境。C 标准需要一个独立的实现来支持这个语言本身的所有特性。然而，对于 C 标准库来说，这样的实现只要提供 4 个标准头文件定义的功能就可以了——<float.h>、<limits.h>、<stdarg.h> 和 <stddef.h>。它也可以支持更多，但是 C 标准没有对中间级别进行说明。

在一些环境中，实现必须提供整个的 C 标准库，这种环境就称为宿主环境，它是用来指代完全实现 C 标准的环境的正式术语。当然，这本书主要关心的是描述这样的宿主环境。假设任意的独立环境想要严格遵循 C 标准，它只需要提供所需的 4 个标准头文件。

这个要求说清楚了 <stddef.h> 中应该包括的东西，因为其他 3 个标准头文件只适用于某些特定的领域：

- <float.h> 描述了浮点数表示的属性；
- <limits.h> 描述了整数表示的属性；
- <stdarg.h> 提供了遍历可变参数表时需要的宏。

独立环境程序使用的所有其他类型或者宏定义都只能由一个头文件提供，即 <stddef.h>。

委员会后来的一个决定使这些头文件变得有点乱，因为有几个类型和宏被多个头文件定义了。例如，头文件 <locale.h> 定义了宏 NULL，而 <stddef.h> 和其他四个标准头文件也定义了。相似地，size_t 和 wchar_t 类型在 <stddef.h> 和其他的标准头文件中都有定义。如果一个头文件总是从其他的地方复制有用信息，那就削弱了这个头文件对定义的作用。不过，其他标准头文件在独立环境中可能无效。

<stddef.h> 中定义的类型和宏另外还有一个共同点，就是它们每一个都可能在某个时候加到语言本身中，因为它们最后可以被翻译器以特有的方式来定义。编写可代替这些定义的可移植代码并不是很容易，甚至有时候根本不可能。

另一方面，作为一个规则，<stddef.h> 中定义的所有的类型和宏都可以作为常规的类型和宏的定义来编写。实现者只需要关心一个具体的翻译器怎样定义某些类型和操作就可以了。

同义类型 考虑一下这个头文件中的 3 个类型定义——ptrdiff_t、size_t 和 wchar_t，每一个都和一个标准整型同义。例如，一个实现不能定义 *short* 为 16 位，wchar_t 为 24 位，而 *int* 为 32 位。它必须让 wchar_t 和指定一个类型定义的某个类型相同，另外两个类型定义也是如此。

宏 NULL 实现宏 NULL 仅仅需要从以下的几个选项中选出最合适的——0、0L 或者 void(*)0。选择一种形式，使得它可以在没有函数原型的情况下作为一个指向 *void* 类型的指针（或者指向 *char*、*signed char* 或 *unsigned char* 类型的指针）类型的参数正确工作。（11.3 节中会更加详细地讨论宏 NULL。）

在 C 语言中包含一个空指针常量可能会更好。这个建议被提出过无数次。然而，其中的一种形式已经可以满足 NULL 可能被使用的情况了。

宏 offsetof 还有一个宏是 offsetof。可以使用这个宏确定一个结构成员和这个结构的起始位置的偏移量（以字节为单位）。标准 C 没有定义编写这个宏的可移植方式，然而，每一个实现肯定会有定义它的某个非标准形式。例如，一个实现可能会可靠地计算某些表示式的值，而这些表达式的行为却不在 C 标准的定义之列。

可以把 offsetof 看作执行一个不可移植操作的可移植方式，这对 C 标准库中的很多宏和类型定义来说都是成立的。实际上适当扩展 C 语言本身的需求并不只是在这几个实例中，这就是头文件 <stddef.h> 存在的原因。

11.2 C 标准的内容

7.1.6 一般定义 <stddef.h>

<stddef.h>

以下的类型和宏由标准头文件 <stddef.h> 定义，某些也有在其他的头文件中定义，在它们相应的条款中都有说明。

类型有如下几个：

ptrdiff_t

ptrdiff_t

是两个指针相减的结果的有符号整数类型。

size_t

size_t

是 sizeof 操作符的结果的无符号整数类型。

wchar_t

wchar_t

是一个整值类型，它范围内的值可以表示最大扩展字符集中所有成员的不同编码值，而该字符集是由支持它的区域设置指定的。空字符的编码值应该为零。5.2.1 中定义的基本字符集的每一个成员的编码值都应该和它在一个整数字符常量中作为单独字符使用时的值相等。

宏有如下几个：

NULL

NULL

展开为实现定义的空指针常量；

offsetof (*type*, *member-designator*)

offsetof

展开为一个 size_t 类型的整值常量表达式，它的值是从结构的起始位置（由 *type* 指定）到结构成员（由 *member-designator* 指定）的偏移量，以字节为单位。*member-designator* 应该满足：

static *type* t;

然后表达式 &(t. *member-designator*) 就会计算一个地址常量。（如果指定的成员是一个位域，则这种行为未定义。）

参见：区域化（7.4）。

11.3 <stddef.h> 的使用

对 <stddef.h> 中的类型定义和宏定义的使用本质上没有什么联系。需要使用该头文件提供的一个或者更多的定义，就包含该头文件。然而，只有 ptrdiff_t 类型定义和 offsetof 宏定义是这个头文件特有的，通常包含另一个标准头文件就会提供需要的定义。下面就分别每一个类型和宏的定义。

11

类型 ptrdiff_t

在一个 C 表达式中对两个指针相减时，结果的类型就是 ptrdiff_t。它是一个可以表示负值的整数类型，因此几乎可以确定它是 *int* 或者 *long* 类型。该类型通常是一个有符号类型，和下面描述的无符号类型 size_t 的位数相同。（上面我说过这些定义的使用本质上是没有联系的，而这两个定义本身是紧密联系在一起的。）

只有当两个指针具有可兼容的数据对象类型时，才能对它们进行相减。例如，其中一个有 *const* 类型限定符，而另一个没有，但它们指向相同的数据对象类型。翻译器可能会检查类型，如果不兼容就会发出警告。通常它不能确定是否有这个附加的限制——这两个指针必须指向同一个数组数据对象的元素。编写不满足这个限制条件的表达式，通常会从这个减法中得到一个没有任何意义的结果。

它的计算方法实际上是按照下面的步骤进行的。程序把这两个指针都表示为对公共地址空间中的某一个点的偏移量（以字节为单位）；然后把这两个偏移量相减，得到一个有符号的中间结果；接着让该中间结果除以这两个指针指向的数据对象的大小（以字节为单位）。如果这两个指针指向同一个数组的元素，那么这个除法就不会产生余数。不论这些元素是什么类型，最后的结果都是这两个数组元素的下标之差。

这就是说，例如，表达式 &a[5]-&a[2] 的值总是 3，类型为 ptrdiff_t。相似地，&a[2]-&a[5] 的值总是 -3。这两种情况中都假设 a 是一个至少有 5 个元素的数组数据对象。（如果 a 恰好含有 5 个元素，对于 &a[5] 这种情况，指针算术就是为一个数组“末尾之后”的元素定义的。）

溢出　在一些实例中，ptrdiff_t 可能不是很完备。假如在一个实现中，size_t 是 *unsigned int* 类型，那么 ptrdiff_t 就是 *int* 类型。然后我们声明一个 *char* 类型的数组数据对象 x，它的大小 N 大于 INT_MAX 字节。（头文件 <limits.h> 把宏 INT_MAX 定义为 *int* 类型可以表示的最大的正值。）然后编写类似于下面的代码：

```
#include <limits.h>
#include <stddef.h>

#define N   INT_MAX+10
.....
    char x[N];
    ptrdiff_t n = &x[N] - &x[0];
```

那个对 n 进行初始化的表达式的结果是什么？因为结果太大而不能表示一个 ptrdiff_t 类型的整数，所以会发生溢出，结果是未定义的。这个问题是不可避免的，它是标准 C 语言所固有的缺点。

虽然存在这样的不足，但这样的情况很少发生。它只可能发生在数组元素只占一个字节的数组身上，而且这些元素是 *char*、*signed char* 或者 *unsigned char* 类型，其他类型时很少发生。它可能发生在那些 *int* 类型是 16 位表示的计算机体系结构下，也可能发生在那些允许创建庞大的数据对象的计算机体系结构下。

即使那样，也只有当指向两个相距了半个以上地址空间的字符数组元素的指针相减的时候才会发生溢出。而且就算发生了溢出，可能也不会造成什么问题，因为 2 的补码算术（现在最常用的形式）也可以纠正很多错误。程序可能会顺利地通过这些危险，并且按照你的意愿去执行。

我叙述所有的这些秘密是为了证明一个简单的结论：我们很少需要使用 ptrdiff_t 类型定义。存储指针减法的结果或者两个下标的差只是我可以想

到的实际应用。通常，程序只会在训练时用到这样的结果。这种类型本身的不足决定了它不可能获得所有的指针减法的结果，这样就限制了它在可移植程序中的使用。知道可以确定一个指针减法的结果的类型就行了，并不用经常关注它。

类型 size_t

在一个 C 表达式中使用 sizeof 运算符的时候，结果的类型就是 size_t。它是一个无符号整数类型，可以表示能声明的最大数据对象的大小，因此几乎可以肯定它是 *unsigned int* 或者 *unsigned long* 类型。它总是一个和上面所讲的有符号类型 ptrdiff_t 具有相同位数的无符号类型。

然而，和 ptrdiff_t 不同的是，size_t 非常有用。它是表示用作数组下标的任意整数数据对象的最安全的类型。使用它就不必担心小的数组会随着程序的变化而演变为很大的数组，而且使用 size_t 时，下标算术永远都不会溢出。也不必担心程序是否能移植到某些具有特殊属性的机器中，例如 32 位的字节和单字节 *long* 类型。类型 size_t 为代码的稳定性提供了最大的保障。计算数据对象大小的唯一可用的类型就是 size_t。

C 标准库广泛使用了 size_t 类型，很多函数的参数和返回值都声明为这种类型。对于那些经常导致程序漏洞的旧的编程习惯来说，这是一种刻意的改变，它抛弃了把所有的整数都声明为 *int* 类型，这也是一种趋势。

在程序中所有对数组下标或者地址进行算术操作的地方，都应该使用 size_t 类型。然而，无符号整数运算比有符号运算的缺陷更多。例如不能使一个无符号计数器递减到一个负数——永远都不会。如果翻译器不对这样的判断表达式发出警告，程序就可能会陷入死循环。事实上，有时候递减到零会使得判断很不灵活。所以，有时候会因为没有使用负值（例如 <stdio.h> 中定义的表示文件结束的 EOF）而不能方便地对它们进行判断，当然，健壮性方面的改进很值得进一步研究。

本书中的代码在所有适合的地方都使用了 size_t 类型。有时候读者可能会在某些地方使用 *int* 数据对象表示下标。然而，在所有这样的情况下，相关的数组数据对象的大小应该很自然地被限制在一个安全的大小范围内。只有当必须将负值和下标混合使用的时候，我才这样做。

类型 wchar_t

写一个宽字节字符的字符串字面量，例如写为 L'x'，它的类型就是 wchar_t；写一个宽字节字符文本字符串，例如写为 L"hello"，它的类型为 wchar_t 数组。wchar_t 是一个整数类型，它可以表示实现支持的所有宽字节字符编码的所有编码值。

对于一个只对宽字节字符提供最小支持的实现来说，wchar_t 可能和 *char* 一样小。对于一个非常大的实现，它可能和 *unsigned long* 一样大。更多情况下，wchar_t 是一个至少具有 16 位表示、像 *short* 或者 *unsigned short* 这样的整数类型。

可以用 wchar_t 来表示所有必须保存宽字节字符的数据对象。<stdlib.h> 中声明的函数中有几个是对宽字节字符进行操作，一次操作一个字符或者以空字符结尾的字符串的一部分这组函数中很多函数的参数和返回值都声明为这种类型。因此，头文件 <stdlib.h> 也定义了 wchar_t 类型。

宏 NULL

宏 NULL 几乎可以作为一个几乎通用的空指针常量使用，可以把它作为不指向程序声明（或者分配）的任何数据对象的数据对象指针的值使用。如 11.1 节提到的那样，这个宏可以是定义 0、0L 或者 (void*)0 中的任意一个。

最后一个定义和所有的数据对象指针兼容，但与函数指针不兼容。这就意味着不能这样写代码：

```
int (*pfun)(void) = NULL;    /*WRONG */
```

翻译器可能会警告表达式类型和要初始化的数据对象的类型不兼容。

NULL 的一个重要的传统用途现在已经不用了。C 语言的早期版本没有函数原型。翻译器不能检查函数调用的参数表达式是否和相应的函数参数声明兼容。因此，它不能对一个可兼容但是类型不同的表达式的表示（例如把 tan(1) 改为 tan(1.0)）进行调整。所以程序员必须保证每一个参数值都有正确的表示。

现代编程风格是对调用的所有函数都要声明函数原型。然而，存在一种重要的情况，当调用一个接受可变参数表的函数（像 <stdio.h> 中声明的 printf）时，函数的参数就没有对应的参数声明。对那些附加的参数，旧的 C 规则也适用。当然，其中会发生一些标准的类型转换。但是大部分时候，每个这样的参数是否正确，是由程序员决定的。

在 C 语言最早的实现中，所有指针的表示都相同。通常，这个表示和整数类型 *int* 或者 *long* 中的一个大小相同。因此，十进制常数 0 或者 0L 中的任何一个都很有可能被误认为是某种类型的空指针。把 NULL 定义为这两个常数中的一个，然后就可以把它赋给一个任意的指针。这个宏作为一个参数表达式的时候特别有用，它说明这个表达式为某种指针类型并且是一个空指针常量。

后来出现了一些实现，它们认为指针和所有整数类型都有很大的不同。编写空指针唯一安全的方式是使用强制类型转换，就像 (char *)0。如果所

有的指针看起来都相同，仍然可以将 NULL 定义为类似（char *）0 这样。这个宏仍然可以作为编写参数表达式的一种有效的方法。

标准 C 允许不同的指针类型有不同的表示。完全可以把任意的数据对象指针转换为指向 *char* 的指针类型（或者指向 *signed char* 的指针，或者是指向 *unsigned char* 的指针），并且可以不用损失任何信息就能把它转换回来。可以方便地把指向 *void* 的指针作为一般的数据对象指针类型使用，特别是声明函数参数和返回值的时候。

在这样的实现中，NULL 最安全的定义是 (void*)0。但是，这不能保证指向 *void* 的指针和任何其他的（非字符）指针有相同的表示，它甚至不能和函数指针赋值兼容。因此，不能把 NULL 编写为一个通用的空指针常量，也不能安全地把它作为一个参数表达式来代替一个任意的数据对象指针。我们只能保证它可以作为一个字符指针或者一个通用的指向 *void* 的指针正确使用。

编写 C 程序的一种现代风格是完全避免使用 NULL。用一个合适的强制类型转换来编写每一个空指针常量，像 (int *)0 这样。虽然这样会增加程序的长度，但是会使程序变得很明确。这种风格的一个改进是在所有需要的地方简单地写一个 0 来作为空指针常量。这对翻译器来已经足够清楚了，但是对代码的阅读者就不同了。

本书使用的风格是尽可能多地使用 NULL。*存在*空指针常量是一个非常有用的信号。我使用强制类型转换生成用于函数指针的空指针常量。我也用它们来表示接受可变参数表的函数参数，特别是当要求的类型不是指向 *void* 的指针的时候。

NULL 在 6 个不同的头文件中都有定义，很容易使用。唯一的建议就是选择一种风格，然后坚持使用它。

宏 offsetof

宏 offsetof 用来确定结构的一个成员距离结构起始位置的偏移量（以字节为单位）。如果想使用一个表驱动函数来对一个结构的单个成员进行操作，这一点很重要。例如，可以参考图 6-15 的函数 _Makeloc 和图 6-13 的表 _Loctab。

这个宏的结果是一个 size_t 类型的整型常量表达式，这意味着可以用它来初始化一个静态数据对象，例如一个具有整型元素的常量表。这也是实现它的唯一可移植的方法，如果编写下面的代码：

```
struct xx {
    int a, b;
    } x;
static size_t off = (char *)&x->b - (char *)&x;
```

则最后一个声明的行为是未定义的。有些实现选择了求取初始化式的值，并得到那个明显的结果；其他的实现则选择了诊断这个表达式。

也不能通过执行指针算术实现从成员到成员的步进，<stdarg.h> 中定义的宏可以实现对一个接受可变参数表的函数执行从参数到参数的步进。那些宏，或者类似于它们的宏，不能保证在一个结构内正常运作，因为结构成员之间的空隙和函数参数之间的空隙不同。实际上，它们不需要遵循任何备有文档的规则。

需要使用宏 offsetof 来编写可移植的代码，如：

```
#include <stddef.h>

struct xx {
    int a, b;
    } x;
static size_t off = offsetof(struct xx, b);
```

11.4 <stddef.h> 的实现

图 11-1 显示了文件 stddef.h，它非常简单。我又一次使用了内部头文件 <yvals.h> 来提供随实现改变的信息。在这种情况下，那些信息确定了所有的 3 个类型定义和宏 NULL 的形式。头文件 <yvals.h> 一般包含以下的定义：

头文件 <yvals.h>

```
typedef int _Ptrdifft;
typedef unsigned int _Sizet;
typedef unsigned short _Wchart;
#define _NULL (void *)0
```

这些定义在很多实现上都可以工作。然而，某些实现可能要求对它们中的一个或者多个进行修改，这就是对它们进行参数化的原因。

宏 offsetof

对宏 offsetof，我使用了一个常用的技巧。很多实现都允许通过强制类型转换把一个整数零转换为一个指针类型，然后对结果执行指针算术。C 标准没有定义这种行为，所以有些实现可能不支持它。

翻译器必须允许这种宏定义，这样才可以正确工作。在 size_t 隐藏起来的时候，翻译器必须通过强制类型转换把一个零开头的（zero_based）地址转换回一个整数类型。而且，它必须容忍整型常量表达式中这样滑稽的行为。这也就是初始化静态数据对象所需要的。

幸运的是，很多翻译器都允许这样做。如果某个实现不允许，就必须考虑实现希望用户怎样定义 offsetof。为了遵循 C 标准，每一个实现都必须提供某种方法。

图 11-1 stddef.h

```
/* stddef.h standard header */
#ifndef _STDDEF
#define _STDDEF
#ifndef _YVALS
#include <yvals.h>
#endif
        /* macros */
#define NULL     _NULL
#define offsetof(T, member)((_Sizet)&((T *)0)->member)
        /* type definitions */
#ifndef _SIZET
#define -SIZET
typedef _Sizet size_t;
#endif
#ifndef _WCHART
#define _WCHART
typedef _Wchart wchar_t;
#endif
typedef _Ptrdifft ptrdiff_t ;
#endif
```
□

11.5 <stddef.h> 的测试

图 11-2 显示了文件 tstddef.c，它检验了 <stddef.h> 中定义的类型和宏的基本属性。该文件是一个简短的程序，因为它提供可供测试的东西非常少。这个程序也显示了 size_t 和 wchar_t 类型的数据对象的大小。(ptrdiff_t 和 size_t 的大小相同。）如果一切顺利，程序显示类似下面的输出：

```
sizeof (size_t) = 4
sizeof (wchar_t) = 2
SUCCESS testing <stddef.h>
```

11.6 参考文献

P. J. Plauger, “Data-Object Types,” *The C Users Journal*, 6, no. 3 (March/April 1988). 这篇文章讨论了和本章的话题相关的几个问题。

11.7 习题

11.1 确定你的实现为 ptrdiff_t、size_t 和 wchar_t 选择的整数类型。

11.2 写一个程序，该程序可以用实验方法确定可以代替 wchar_t 的整数类型。

11.3 编写一个程序，该程序可以用实验方法确定可以代替 ptrdiff_t 和 wchar_t 的整数类型。

11

图 11-2
tstddef.c

```
/* test stddef definitions */
#include <assert.h>
#include <limits.h>
#include <stddef.h>
#include <stdio.h>

        /* type definitions */
typedef struct {
    char f1;
    struct {
        float flt;
        } f2;
    int f3;
    } Str;

        /* static data */
static char *pc = NULL;
static double *pd = NULL;
static size_t offs[] = {
    offsetof(Str, f1),
    offsetof(Str, f2),
    offsetof(Str, f3)};

int main()
    {                   /* test basic workings of stddef definitions */
    ptrdiff_t pd = &pc[INT_MAX] - &pc[0];
    wchar_t wc = L'Z';
    Str x = {1, 2, 3};
    char *ps = (char *)&x;

    assert(sizeof (ptrdiff_t) == sizeof (size_t));
    assert(sizeof (size_t) == sizeof (sizeof (char)));
    assert(pd == &pc[INT_MAX] - &pc[0]);
    assert(wc == L'Z');
    assert(offs[0] < offs[1]);
    assert(offs[1] < offs[2]);
    assert(*(char *)(ps + offs [0]) == 1);
    assert(*(float *)(ps + offs[1] ) == 2);
    assert(*(int *)(ps + offs[2]) == 3);
    printf("sizeof (size_t) = %u\n", sizeof (size_t));
    printf("sizeof (wchar_t) = %u\n", sizeof (wchar_t));
    puts("SUCCESS testing <stddef.h>");
    return (0);
    }
```
□

11.4 [**难**] 某些实现允许对一个整型常量表达式中的两个指针相减，前提是它们都以某个静态数据对象声明为起点，写出一个可以利用这种功能的 offsetof 的定义。

11.5 [**很难**] 把一个空指针常量添加到 C 语言中。关键字 nul 是一个和所有的指针类型兼容的空指针。如果没有相应的参数声明，如何把 nul 作为参数表达式处理？

第12章

<stdio.h>

12.1 背景知识

头文件 <stdio.h> 声明了很多执行输入输出的函数。几乎所有的程序都要执行输出操作，所以这个头文件被广泛使用。事实上，它是最早出现在C标准库中的头文件之一。这个头文件比其他任何头文件声明的函数都要多。同时由于实现这些函数机制很复杂，因此它也需要更多的说明。

本章主要讨论以下几个话题：

- C标准库实现的抽象输入/输出模块；
- 读出和写入原始数据的低级函数；
- 在格式规范控制下打印和扫描数据的高级函数。

我们从历史的角度开始讨论。

输入/输出模型

在过去的几十年里，计算机编程的某个领域得到了飞速的发展，但它的成功并没有得到足够的认可。我指的是在过去的二十年左右和高级语言一起得到提高的独立于设备的输入输出模型。标准C从这个改善的模型中获益颇丰。

在20世纪60年代早期，FORTRAN IV被认为是独立于机器的语言。但是如果不作任何改动，根本不可能在各种计算机体系结构间移动FORTRAN IV程序。可移植性的主要障碍是输入输出（简写为I/O）领域。在FORTRAN IV中，可以对FORTRAN IV代码中间的I/O语句中对正在通信的设备进行命名。例如，在面向磁带的机器IBM 7090上读输入卡片时，就用`READ INPUT TAPE 5`，但是在其他的机器上就用`READ CARD`。打印结果时，使用`WRITE OUTPUT TAPE 6`、`PRINT`或者`TYPE`。

逻辑单元数

后来FORTRAN IV出现了并提供了一个出路。现在可以编写更多通用的`READ`和`WRITE`语句，每一条语句都指定了*逻辑单元数*（或者LUN）来代替具体的设备名。这样就必须在可执行的二进制卡片前加入控制卡片，这样就可以指定在这个特殊的运行过程中哪些设备和哪些LUNs相对应。独立于设备的I/O时代来临了。

当然，外围设备很清楚地知道用户想要它们做的事情。例如，当向打印机输入字符时，每一行的第一个字符被转化为控制回车空格。把相同的行送向打字机，就会打印出回车控制字符。与磁带和磁盘文件的阻塞问题、二进制卡片格式或者怎样对各种输入指定文件结束相比，回车控制只是一个很小的问题。很快，我们就能学会可以选择哪些成对的设备来进行某种输入输出了。

PIP 功能

设备独立的进一步改善得益于标准外围交换程序（peripheral interchange programn，简称 PIP）的进步，该程序允许指定源设备与目标设备的任意组成，然后尽力执行两个设备间的拷贝操作。通常情况下，必须指定一些特殊的选项来使 PIP 尽可能选择正确。但是不论提供了多少提示，有时候还是不能得到某些想要的组合。

后来 CRT 终端产生了，然后每个人都可以单步地退回。结束一行是用回车还是回车加上换行，或者是换行符，还是某些更加神奇的字符？终端允许水平制表符号设置和展开制表符吗？还是不允许使用制表符？怎样用键盘标志文件结束呢？这些问题的答案几乎和 CRT 终端的厂商一样多。

进入 UNIX 时代

20 世纪 70 年代早期出现了 UNIX。该著名系统的开发者 Ken Thompson 和 Dennis Ritchie，由于为 UNIX 注入了很多好的思想而得到了应得的赞扬。他们设计的关于设备独立的方法是最好的方法之一。

UNIX 对所有的文本流都采用了标准内部形式，文本的每一行以换行符终止。这正是程序读入文本时所期望的，也是程序输出时所产生的。假如这样的约定不能满足和 UNIX 机器相连的处理文本的外围设备的需要，可以在系统的对外接口作些修改，不必修改任何内部代码。

系统调用 `ioctl`

UNIX 提供了两种机制来修正“对外接口”的文本流。首选的机制是一个通用的映射函数，它可以用任意的文本处理设备工作。可以用系统调用 `ioctl` 来设置或者测试一个具体设备的各种参数。使用 `ioctl` 就可以（在其他事情中）对内部换行约定和各种终端的需要之间的各种转换进行选择。近年来，`ioctl` 已经演变为一个非常成熟的小 PIP，它用来控制文本处理设备。

设备管理器

另一个修正文本流的机制是修改直接控制该设备的专门软件。对每一个 UNIX 可能需要控制的设备来说，用户必须添加一个常驻 UNIX 的设备管理器。（MS-DOS 采用了类似的机制。）早期，Thompson 和 Ritchie 发布了一个先例，就是每个设备都应该在所有需要的场合处理标准文本流。

文件描述符

当 Dennis Ritchie 使第一个 C 编译器在 PDP-11 UNIX 平台上运行时，C 语言就自然地继承了它的宿主操作系统简单的 I/O 模型。除了文本流的统一表示，还有其他一些优点。很久以前使用的 LUNs 在最近几十年也慢慢地演变为称为*文件描述符*或*句柄*的非常小的正整数。操作系统负责分发文件描述符，并且把所有的文件控制信息存储在自己的专用内存中，而不是让用户去分配和维持文件和记录控制块以加重负担。

为了简化多数程序的运行管理，UNIX shell 分配给每个运行的程序 3 个标准文件描述符，这些就是现在普遍使用的*标准输入*、*标准输出*和*标准错误流*。（严格地说，它们不是 UNIX 的发明，因为它们在 PL/I 和 MULTICS 等操作系统中就已经出现了。）程序员迅速学会了在所有可能的情况下都从标准输入流读取文本并输出文本到标准错误流。软件工具也因此而产生。

二进制流

另外一个很小但是很重要的改进是 8 位透明性。UNIX 并不会阻止向任意打开的文件中写入任意的二进制编码，或者从一个足够大的地方把它们丝毫不变地读取出来。确实，把二进制编码写入一个文本处理设备可能会产生很多奇怪的结果，但是一个文件或者管道通常可以并且愿意存储任意的二进制流。程序员最终学会了让程序容纳任意的二进制编码，无论什么时候这都是有必要的，即使这些程序是作为文本处理工具而开发的。所以，UNIX 消除了文本流（与人通信）和二进制流（与程序通信）之间的区别。

文件长度

还有一点改进是精确长度的文件。大部分操作系统都尝试过将保存在磁盘、磁带或者其他存储设备上的文件的底层块结构隐藏起来，但很多都半途而废。当把数据写到文件中，然后再读回来的时候，可能会有 1 到 1000 个额外的字符被添加到了文件的末尾。UNIX 用最接近的字节数来记录文件的大小，所以最后得到的就是写到文件中的内容。设备处理器的程序员通常都知道提供一种机制，这种机制可以保证出入设备的数据流的整洁性。因此它又回到了对曾经风靡一时的 PIP 的需求上。（注意，尽管如此，UNIX 仍然使用 dd 命令——现代的 PIP。）

相似地，创建临时文件不需要提前作准备，甚至不需任何思考。大部分情况是通过管道将不同作者编写的 C 程序连到一起进行工作。那些分发给各个大学的早期的 UNIX 系统培养了一批程序员，他们不用考虑实际的输入输出的各种糟糕的情况，而在当时其他的操作系统下则不是这样。

C 的迁移

当 C 从 UNIX 转移到其他操作系统的时候，就没有这么好的事了。我们处理最早的实现时，遇到了一些棘手的决定。我们应该保留 C 程序员已经习惯的简单的 I/O 模块，还是修改 I/O 库来适应本地习惯？至少从哲学上讲，

这个问题很简单。几乎没有 C 程序员希望在打开文件时操作文件控制块或者指定无数的参数——更不用说多年来简单的 I/O 了。大多数人都选择了尽可能多地保留简单的 I/O 模块。（我们也为那些习惯本地操作系统的人提供了一些能够吸引他们的东西。）

隐藏实现细节

问题是我们把那些繁琐的细节隐藏在什么地方？UNIX 把它们大部分封装在 ioctl 和设备处理程序中。一般来说，我们没有那个选项。因此，我们必须构建更多复杂的库来处理各种设备和表示文本的不同习惯。要保证 C 可以读取和写入那些和本地文本编辑器兼容的文本文件，这一点很重要。C 也必须至少可以从键盘读取文本并将其写入显示设备和打印机。需要的时候，库要在内部的以换行符终止的文本行和外部的本地用法之间进行转换。

在非 UNIX 系统下，我们不能把这些繁琐的东西全部隐藏起来。因此，我们实现者又面临的一个决定是，当我们不能把它们隐藏起来的时候如何利用它们。那些厂商通常只在一种环境下实现 C 库，他们就在现有的 I/O 函数中添加了很多复杂的东西，并且添加了很多新函数。而我们的目标则是在多个系统之间构建统一而且强大的环境，所以我们必须更加谨慎。这就意味着要在现有的 I/O 函数中添加最小的复杂性，并且尽可能少地添加新的函数。同时也意味着要削弱 UNIX I/O 模块的某些承诺来满足各种系统之间那些不通用的命名。

X3J11 开发 C 标准

X3J11 委员会在 1983 年开始召开会议为 C 起草 ANSI 标准。非 UNIX 系统的 C 厂商和那些 UNIX 用户之间争论了很长时间，因为 UNIX 用户不能理解 I/O 为什么要这么麻烦。这是一个很有教育意义的过程。这些争论的一个重要的副产品就是更清楚地阐明了 C 支持的 I/O 模块。

文本与二进制

开始的时候，标准 C 不得不再次引入文本文件和二进制文件之间的差别，因为 UNIX 之外的大部分操作系统都迫使实现区别对待这两种文件。例如，MS-DOS 对文本文件和二进制文件允许使用相同的系统调用，但是在文本文件中结束每一行的时候它都使用回车和换行。C 读取文本文件的时候必须丢弃回车符，但是读取二进制文件的时候又不是这样。因此，即使我们认为不应该有区别，这种区别也是实际存在的。

打开文件的时候可以指定一个文件是作为文本文件还是二进制文件进行处理。例如，可以用 fopen(fname,"r") 来打开一个文件供程序读取，在标准 C 中，这种形式默认为打开文本文件。如果要打开一个 binary 文件，可以用 fopen(fname,"rb")。也可以把 b 加到其他任意模式下。（b 放在模式中 + 的前面和后面都是允许的。）

在 UNIX 和其他不区别这两种形式的系统下，可以忽略 b 模式限定符。然而，在很多系统下，这种区别非常重要。如果程序要移植到其他系统，就要考虑文件是怎么使用的，然后正确地编写 fopen 模式。否则，程序可能会在各种细节方面出错。

要设计一个文本文件使它严格支持文本流的 UNIX 模式并不总是那么容易的。就像本节前面讲 PIP 时提到的那样，文本行结束就有很多种方式。因此，实现要能自由地把外部各种文件格式正确地转换为程序所能读的格式，也能把程序所写的内容转换为外部其他程序可以使用的形式。这个自由要包括写到文本文件的字符集、如何构建文件行，甚至零和空之间的区别。

某些系统在向文本文件输出的时候不支持 8 位透明性。ctl-Z 在很多流行的操作系统下都像是文件结束符。甚至 C 基本字符集中的字符也是不确定的。某些环境可能不支持换页符和垂直制表符。实际上，为了程序的最大可移植性，写入文本文件时应该只使用可打印字符，加上空格、换行和水平制表符。

结束文本行 如果最后一行文本不是一整行，很多系统都不能很好地处理它们，因为没有结束符它们就不能表示行的概念。如果写入文本文件的最后一个字符不是换行符，那么最后的不完整的文本行就可能丢失；或者自动补充完整，因而再次读取的文本就多了一个换行符；或者程序运行的时候会出现问题。所以要避免文本文件的最后一行不完整。

文本行长度 某些系统甚至不能表示空文本行。因此输出空文本行的时候，库实际上输出的可能是只包含一个空格的文本行。在读取的时候，系统就会丢弃只包含一个空格的文本行中的空格。还有一些系统丢弃文本行后面的所有空格。因此，如果程序读取一个包含了固定长度的文本记录的文件时，就会方便很多，所有尾部空格都消失了。但是，这就意味着不能指望在输出尾部有空格的文本之后再把那些空格读回来。在可移植程序中不要这样做。

另一个极端是，系统可以设置它们能读取或者输出的最长的文本行长度的上限。高于上限的那些字符可能会被删除，所以尾部的字符会丢失。或者它们被折叠，然后就可能突然遇到原本不存在的换行符。或者程序运行的时候会报告错误。C 标准保证的文本行长度的上限是 254 个字符。（理论上，在处理过反斜线连接之后，最长的 C 源文件行是 509 个字符。）

文件长度 某些系统不能表示空文件。如果创建了一个新文件但没有写入任何东西，然后把它关闭，系统就没办法将空文件与不存在的文件区分开来。因此，标准 C 允许实现在关闭空文件的时候删除它们。一定要注意这一点。

另外，一个非常大的文件也可能出现问题。在 UNIX 下，可以用一个 32 位的整数表示文件中任意字节的位置。C 传统的文件定位函数认为 *long* 类型可以表示任意的文件位置，但在其他的系统下并非如此，即使文件的长度小于 2^{32} 个字节。因此，委员会向 C 标准库中添加了一个可供选择的文件定位函数集来部分改善这个问题。

在结束对文本文件的正面讨论之前，我提供一点鼓励的话。只要遵守所有的规则，那么程序写入文本文件的字符序列和最后从文本文件中读出的字符序列就可以完全相同。如果程序有这样的要求，就不要存在侥幸的心理弯曲这些规则。

二进制文件 对二进制文件来说所作的妥协是重新引入长度的不确定性。实现必须精确地保留程序写入文件开头的所有字节，但可以自由填充二进制文件。只要填充字符的值为零（'\0'），就可以对其添加任意数目。所以设计二进制文件的时候要更加小心。不要认为在读取了写到文件中的最后一个字符之后就能看到文件结束符。或者通过某种方式得到数据结束的标志，或者忍受读取数据的尾部的多个零字节。

流的演变 正如本节前面所提到的，UNIX I/O 在早期的系统下表示非常简化。UNIX 之前设计的大部分系统都认为 I/O 理应是一个很复杂的操作，它的复杂性不能在执行的程序中隐藏。文件有各种各样的结构，反映为它的各种属性，例如块或者记录的大小、查找关键字、打印格式控制等等很多种。那些执行 I/O 的系统调用必须指定这些属性的不同组合。而且其他信息位还必须在各个系统调用之间保留，以记录每一个流的状态。

所以，最简单的事情似乎是系统要求每一个用户程序分配存储空间，以此传递或者 / 和记录所有这些属性值和其他的状态信息位。这个存储区域称为“数据控制块”、“文件控制块”、“记录访问块”或者其他同样模糊的名字。在打开一个文件之前，必须为控制块开辟一块空间，把指向控制块的指针传递给打开文件的系统调用，然后把该指针传递给所有对这个文件进行 I/O 操作的后继系统调用。I/O 系统调用需要的其他的参数全部放到控制块的各个域中。

如果幸运的话，操作系统的厂商会提供一个汇编语言宏集合来分配这些控制块并对各种域进行编址。用户要谨慎地使用这些宏，因为大部分厂商都认为用户可以自由地改变控制块的大小和布局。宏的接口很稳定，因为如果它改变的话，会给厂商的系统程序员造成不便。

即使是世界上最好的宏包，用户仍然要学习使用它的非结构化的接口。作为一个规则，汇编语言几乎不能保证从控制块的域中读取或者写入适当类型的数据。更糟的是，这些域非常多而且有可能相关文献记录得不太好。所有用户不能确定是否要在某个系统调用之前设置某些域，或者在系统调用之后使用某些域来包含有意义的信息。唯一确定的是，控制块中的错误修改可能会冻结 I/O、损坏文件，甚至使系统崩溃。

UNIX I/O 模型

所以当 UNIX 不再需要用户存储区的控制块时，才是真正的进步。在 UNIX 打开一个文件时，用户只获得一个文件描述符，它是一个小正整数。所有的控制信息都保留在系统中，这样基本可以避免用户程序的误操作。由于文件是非结构化的，所以每次 I/O 系统调用时都要指定一些参数。把 C 调用的函数使用的标量参数跟任何具体的实现下 UNIX 系统调用要求的最小的（和暂时的）结构对应起来是非常简单的。

执行 UNIX 风格的 I/O 的 C 函数有 `open`、`close`、`read`、`write` 和 `lseek` 等，它们在文件描述符和 I/O 缓冲区中通信。它们支持简单的 I/O 模块，这些模块被强制使用在几十个更加复杂的操作系统下。似乎，它们是标准 C 下 I/O 原语的理想候选。

然而，存在一个小问题。当最早为 UNIX 编写的程序直接调用这些原语时，后面的程序就变得更加复杂。它们在用户存储区强制使用了一个缓冲层，以此来使出入程序的数据的单位字节所使用的系统调用数目最少。程序每次系统调用时读取和写出数百个字节几乎总是比一些字节的速度快很多。

然后又出现一个标准库，这个库中的函数就自动承担起了分配和释放缓冲区，填充和提取它们，并用统一的形式跟踪错误情况等责任。这些函数工作时使用指向控制流的 `FILE` 类型的结构。每一个流数据对象都要记录相关文件的 I/O 状态。它也包含了一个指向缓冲区的指针和附加的状态信息，这样就可以记录缓冲区中有用字节的数目。

选择 I/O 原语

然后 X3J11 委员会一致通过了把流添加到 C 标准库中的决定。很多人学习专门使用流来工作以保证良好的 I/O 性能。甚至一些 C 实现选择了专门实现 I/O 流，而鄙弃了更简单的 UNIX 风格的原语，因为它的效率太低了。

某些以 UNIX 原语为基础的实现在调用 `read` 和 `write` 的时候，经常不得不在用户存储区缓冲数据，要是能在结构化文件中对数据进行压缩和解压

缩就好了。使用流函数的用户总是遭受两层缓冲的折磨，它不仅没有改善性能，而且经常使那些交互式程序出现问题。

所以出现了进退两难的局面：要提高程序的性能必须在流级别执行 I/O，甚至在 UNIX 下也是如此。可以只根据几个面向流的函数，例如 fopen、fclose、fgetc、fputc、fgetpos 和 fsetpos 等来定义所有的 I/O。然而，这样做就忽略了那个广泛的历史事实，就是也可以使用更简单的 UNIX 风格的原语实现 I/O。这样就不用使用 FILE 数据对象和用户空间中分配的缓冲了。在非常小的系统上用 C 编程的人希望可以节省额外的空间，哪怕以性能为代价也可以。

然而，从一个标准的立场看，不得人心的一个地方就是，为了达到一个共同的目标而存在两种不同的机制。委员会经过讨论整洁的重要性和向下兼容的重要性之后，决定抛弃 UNIX 风格的原语。

最后，这个争论让大多数人相信，一个实现总应该添加 open、close 等函数作为扩展。当然，这些函数不能和用户定义的同名的函数或者数据对象冲突。这意味着实现也可以不提供这些函数。这又意味着类似 fopen 一定不能调用 open。而且，提供传统的 UNIX I/O 原语也可以不违背 C 标准。

委员会中的某些实现者甚至认为可以根据 fgetc 实现 read，且这种方式和其他的方式一样有效，或者效率更高。就像高能物理中的基本粒子一样，知道只有一些函数是原语，但是却很难区分哪些函数是原语，哪些函数是在其他函数基础上构建起来的。

FILE 类型

实际上，标准 C 要求流执行 I/O 是向后退了一步。现在每一个程序都要在用户存储区存放一个复杂的控制块以记录每一个流的状态。分配和释放控制块（FILE 数据对象）的时候一定要很小心。不能直接对控制块和它控制的缓冲进行读取或写入。必须按照某种顺序调用函数来执行流的 I/O 操作。

然而，现在比过去好多了。调用 fopen 打开一个流的时候（或者程序开始之前打开 3 个标准流的时候），系统就分配一个 FILE 数据对象。用户不必知道 FILE 数据对象的内部结构，因为用户永远都不会直接向它传递参数或者从它那里得到某个返回值。C 标准库提供了控制流所使用的读取和写入参数的函数，并且 I/O 函数的语义要求流的行为要很健壮，甚至当用户对它们做一些愚蠢的事情时也是如此。

12.2 C 标准的内容

<stdio.h>

7.9 输入 / 输出 <stdio.h>

7.9.1 简介

头文件 <stdio.h> 声明了 3 种类型、一些宏和很多执行输入输出的函数。

size_t

声明的类型有 size_t（7.1.6 中有描述）；

FILE

FILE

它是一个对象类型，可以记录控制流需要的所有信息，包括它的文件定位符、指向相关缓冲（如果有的话）的指针、记录是否发生了读 / 写错误的错误指示符和记录文件是否结束的文件结束符；

fpos_t

fpos_t

它是一个对象类型，可以唯一指定文件中的每一个位置所需的所有信息。

NULL

声明的宏有 NULL（7.1.6 中有描述）；

_IOFBF
_IOLBF
_IONBF

_IOFBF
_IOLBF
_IONBF

它们展开为具有不同值的整值常量表达式，适合作为函数 setvbuf 的第三个参数使用；

BUFSIZ

BUFSIZ

它展开为一个整值常量表达式，是指 setbuf 函数使用的缓冲的大小；

EOF

EOF

它展开为一个负的整值常量表达式，该表达式由几个函数返回来说明文件的结束，即一个流输入结束了；

FOPEN_MAX

FOPEN_MAX

它展开为一个整值常量表达式，表示实现支持的可以同时打开的文件数目的最小。

FILENAME_MAX

FILENAME_MAX

它展开为一个整值常量表达式，表示一个 char 类型数组的大小，这个数组可以保存实现可以打开的最长的文件名串 [110]。

L_tmpnam

L_tmpnam

它展开为一个整值常量表达式，表示一个 char 类型数组的大小，这个数组可以保存 tmpnam 函数生成的临时文件名串。

SEEK_CUR
SEEK_END
SEEK_SET

SEEK_CUR
SEEK_END
SEEK_SET

它们展开为具有不同值的整值常量表达式，适合作为 fseek 函数的第三个参数使用；

TMP_MAX

TMP_MAX

它展开为一个整值常量表达式，表示 tmpnam 函数可以生成的单独文件名的最小数目。

stderr
stdin
stdout

stderr
stdin
stdout

它们是“指向 FILE 的指针”类型的表达式，分别指向与标准错误流、标准输入流和标准输出流相关的 FILE 对象。

参见：文件（7.9.3）、函数 fseek（7.9.9.2）、流（7.9.2）和函数 tmpnam（7.9.4.4）。

7.9.2 流

流

不管是读出还是写入物理设备，例如终端和磁带驱动器，或者不管是读出还是写入到结构化存储设备支持的文件，输入和输出都和逻辑数据流相对应，这些逻辑数据流的属性比各种各样的输入输出更加统一。C 语言支持两种形式的映射，文本流和二进制流[111]。

文本流

文本流是组成文本行的有序字符序列，每一行都由零个或者多个字符加上一个标志结束的换行符组成。最后一行是否需要结束的换行符是由实现定义的。输入输出时可能必须添加、修改或者删除一些字符来和宿主环境中表示文本的不同约定一致。因此，流中的字符和外部表示的字符之间不必有一对一的对应关系。只有当数据仅由可打印字符、控制字符水平制表符和换行符组成，紧靠换行符前面没有空格，并且最后一个字符是换行符时，从文本流中读出的数据才有必要和前面写入到该流中的数据一致。换行符正前面的空格在读取的时候是否出现是由实现定义的。

二进制流

二进制流是字符的有序序列，它可以透明地记录内部数据。在相同的实现下，从一个二进制流读入的数据应该和之前写入到这个流的数据相同。然而，这样的流可能会有一定数目的空字符附加在流的结束处，具体数目由实现定义。

环境限制

实现支持的文本文件的每一行应该至少可以包含 254 个字符，包括结束的换行符。宏 BUFSIZ 的值至少应该为 256。

7.9.3 文件

打开文件

一个流通过打开一个文件来关联一个外部文件（也可以是一个物理设备），也可能涉及打开一个新的文件。创建一个现有的文件会丢弃这个文件以前的内容。如果文件支持定位要求（例如一个磁盘文件，和终端相对），那么和流相关的文件定位符[112]就定位在文件的起始位置（零号字符处），除非文件用附加模式打开，这种模式下定位符的初始化在文件的起始位置还是结束位置是由实现定义的。文件定位符通过后来的读、写和定位要求得到维护，这样有助于文件的顺序行进。所有的输入都像是通过对函数 fgetc 的连续调用来读取，所有的输出都像是通过对函数 fputc 的连续调用来写入。

除了 7.9.5.3 中定义的那些，二进制文件不会被截短。对一个文本流进行写操作是否会导致相关文件中超出操作点的内容被删除是由实现定义的。

缓冲文件

当一个流无缓冲时，字符很快从字符源或者在目标设备处出现。否则，字符就会累积然后作为一个块被传送到宿主环境或从宿主环境移出。当流为完全缓冲且缓冲区被填满时，字符就会作为一个块传送到宿主环境或者从宿主环境中移出。当一个流是行缓冲时，遇到换行符时字符就会作为一个块传送到或者移出宿主环境。而且，当缓冲区被填满时，且当请求对无缓冲的流输入或者当请求对需要从宿主环境传递字符的行缓冲的流输入时，字符就要作为一个块传送到宿主环境中。对这些属性的支持是由实现定义的，而且可能会受到函数 setbuf 和 setvbuf 的影响。

关闭文件

一个文件可以通过关闭文件来和控制流分开。输出流在流和文件分离之前被清空（任何没有写出的缓冲内容都被传送到宿主环境）。在相关的文件关闭之后（包括标准文本流），指向 FILE 对象的指针的值是不确定的。长度为零的文件（输出流没有向这个文件写入任何内容）是否存在是由实现定义的。

再次打开文件

文件以后可以被再次打开，被同一个或者另一个程序打开均可，它的内容也可以被使用和修改（如果能重新定位到它的起始位置的话）。如果 main 函数返回它的原始调用者，或者调用了 exit 函数，那么在程序结束之前所有打开的文件就都关闭了（因此所有的输出流都被清空了）。程序终止的其他方法，例如调用 abort 函数，就不需要正确关闭所有的文件。

用来控制流的 FILE 对象的地址可能会很重要。不必使用 FILE 对象的副本来代替原

始的对象进行服务。

程序开始执行时已经预定义了 3 个文本流，它们不需要显示打开——标准输入（读入常规的输入）、标准输出（写出常规的输出）和标准错误（写出诊断信息）。标准错误流打开的时候没有被完全缓冲；当且仅当流不会涉及交互式设备时，标准输入和输出流才被完全地缓冲。

打开其他文件（非临时的）的函数需要一个文件名，这个名字是一个串。构成有效的文件名的规则是由实现定义的。同一个文件能否被同时打开多次也是由实现定义的。

环境限制

宏 FOPEN_MAX 的值至少是 8，包括 3 个标准文本流。

参见： 函数 exit（7.10.4.3）、函数 fgetc（7.9.7.1）、函数 fopen（7.9.5.3）、函数 fputc（7.9.7.3）、函数 setbuf（7.9.5.5）和函数 setvbuf（7.9.5.6）。

7.9.4 对文件的操作

remove

7.9.4.1 函数 remove

概述

```
#include <stdio.h>
int remove(const char *filename);
```

说明

函数 remove 会导致一个文件再也不能通过它的文件名进行访问，这个文件的文件名是由 filename 指向的串。后面通过该文件名来打开这个文件的尝试就会失败，除非它被重新创建。如果要删除的文件已经打开了，remove 函数的行为是由实现定义的。

返回值

函数 remove 在操作成功的时候返回零，否则返回非零。

rename

7.9.4.2 函数 rename

概述

```
#include <stdio.h>
int rename(const char *old, const char *new);
```

说明

函数 rename 把名字为 old 指向的串的文件改名为 new 指向的串，这个文件再也不能通过以前的名字进行访问。如果调用函数之前存在一个名为 new 指向的串的文件，函数的行为是由实现定义的。

返回值

如果操作成功，函数 rename 返回零；失败[113]，则返回非零。在新名字的文件存在的情况下，这个文件的名字仍然是原始的名字。

tmpfile

7.9.4.3 函数 tmpfile

概述

```
#include <stdio. h>
FILE *tmpfile(void);
```

说明

- 函数 tmpfile 创建一个临时的二进制文件，当这个文件关闭或者程序终止的时候它会被自动地移除。如果程序异常终止，打开的临时文件是否被移除是由实现定义的。这个文件打开时通过模式 "wb+" 进行更新。

返回值

函数 tmpfile 返回一个指向它创建的文件流的指针。如果不能创建文件，函数返回一个空指针。

参见： 函数 fopen（7.9.5.3）。

tmpnam

7.9.4.4 函数 tmpnam

概述

```
#include <stdio.h>
char *tmpnam(char *s);
```

说明

函数 tmpnam 生成一个字符，这个字符是一个有效的文件名，并且它不和现有的文件名相同。

每调用函数 tmpnam 一次，它就生成一个不同的名字，最多可以调用 TMP_MAX 次。如果调用的次数超过了 TMP_MAX 次，函数行为是由实现定义的。

实现应假设库函数没有调用函数 tmpnam。

返回值

如果参数是一个空指针，函数 tmpnam 把它的结果保留在一个内部静态对象中，并返回指向该对象的指针。对 tmpnam 函数的后续调用可能会修改该对象。如果参数不是一个空指针，就认为它指向一个至少有 L_tmpnam 个字符元素的数组，函数 tmpnam 把结果写到该数组中并返回参数值。

环境限制

宏 TMP_MAX 的值至少为 25。

7.9.5 文件访问函数

fclose

7.9.5.1 函数 fclose

概述

```
#include <stdio.h>
int fclose(FILE *stream);
```

说明

函数 fclose 使 stream 指向的流被清空，并且和流相关联的文件被关闭。这个流中任何未写出的缓冲数据都传递到宿主环境中，然后写入文件中；任何未读出的缓冲数据都被丢弃。流和文件之间的关联也被解除。如果相关的缓冲是自动分配的，就把它释放。

返回值

如果流被成功关闭，函数 fclose 返回零；如果检测到任何错误，就返回 EOF。

fflush

7.9.5.2 函数 fflush

概述

```
#include <stdio.h>
int fflush(FILE *stream);
```

说明

如果 stream 指向一个输出流或者修改流，在这个流中，最近的操作不是输入，函数 fflush 会导致这个流所有未写入的数据传递给宿主环境，然后写入文件；否则，函数的行为是未定义的。

如果 stream 是空指针，函数 fflush 对所有的流执行上述定义的清空行为。

返回值

如果发生写入错误，函数 fflush 返回 EOF；否则，返回零。

参见：函数 fopen（7.9.5.3）和函数 ungetc（7.9.7.11）。

fopen

7.9.5.3 函数 fopen

概述

```
#include <stdio.h>
FILE *fopen(const char *filename, const char *mode);
```

说明

函数 fopen 打开名字为 filename 指向的串的文件，并把这个文件和一个流相关联。

参数 mode 指向以下面列出的字符开头的字符串[115]。

r	打开现有文本文件以便读取。
w	生成新文本文件或截短现有文本文件至零长度以便写入。
a	附加。生成新文本文件或打开现有文本文件以便在文件结束处写入。
rb	打开现有二进制文件以便读取。
wb	生成新二进制文件或将现有二进制文件截至零长度以便写入。
ab	附加。生成新二进制文件或打开现有二进制文件以便在文件结束处写入。
r+	打开现有文本文件，以便更新（读和写）。
w+	生成新文本文件或将现有文件截至零长度以便更新。
a+	附加。生成新文本文件或打开现有文件以便更新，在文件结束处写入。
r+b 或 rb+	打开现有二进制文件以便更新（读和写）。
w+b 或 wb+	生成新二进制文件或截短现有文件至零长度以便更新。
a+b 或 ab+	附加。生成新二进制文件或截短现有文件至零长度以便更新，在文件结束处写入。

用读模式（'r' 作为 mode 参数的第一个字符）打开一个文件时，如果文件不存在或者不能读，操作就会失败。

用附加模式（'a' 作为 mode 参数的第一个字符）打开一个文件会使后面所有对文件的写操作都强制到当前的文件结束处，不管中间是否调用了函数 fseek。在某些实现下，用附加模式（'b' 作为以上 mode 参数值列表的第二个或者第三个参数）打开一个二进制文件可能会把流的文件定位符放置到超过写入的最后一个数据的地方，因为存在空字符填充。

当一个文件用更新模式（'+' 作为以上 mode 参数列表的第二个或者第三个字符）打开时，对相关的流可以执行输入和输出操作。然而，如果中间没有调用函数 fflush 或者文件定位函数（fseek、fsetpos 或者 rewind），输出可能不会直接跟在输入后面；如果中间没有调用文件定位函数，输入也可能不会直接跟在输出后面，除非输入操作恰好发生在文件结束处。在某些实现下，用更新模式打开（或者创建）一个文本文件可能会代替打开（或者创建）一个二进制流。

当一个流打开时，当且仅当可以确定这个流不会涉及一个交互式设备时，它才可以完全缓冲。该流的错误指示符和文件结束符会被清除。

返回值

函数 fopen 返回指向控制流的对象的指针。如果打开操作失败，fopen 返回空指针。

参见：文件定位函数（7.9.9）。

freopen

7.9.5.4 函数 freopen

概述

```
#include <stdio.h>
FILE *freopen(const char *filename, const char *mode,
    FILE *stream);
```

说明

函数 freopen 打开名字为 filename 指向的串的文件，并且把它和 stream 指向的流关联在一起。对 mode 参数的使用与函数 fopen 中的相同[116]。

函数 freopen 首先尝试关闭和指定的流关联的任意文件。如果不能成功关闭，就忽略这一点。然后清空流的错误指示符和文件结束符。

返回值

如果打开文件操作失败，函数 freopen 就返回一个空指针；否则 freopen 返回 stream 的值。

setbuf

7.9.5.5 函数 setbuf

概述

```
#include <stdio.h>
void setbuf (FILE *stream, char *buf);
```

说明

除了没有返回值之外，函数 setbuf 就等价于函数 setvbuf，而该函数是以参数 mode 为 _IOFBF 的值和参数 size 为 BUFSIZ 的值被调用的，或者（如果 buf 是一个空指针）是以参数 mode 为 _IONBF 的值被调用的。

返回值

函数 setbuf 没有返回值。

参见： 函数 setvbuf（7.9.5.6）。

setvbuf

7.9.5.6 函数 setvbuf

概述

```
#include <stdio.h>
int setvbuf(FILE *stream, char *buf, int mode,
      size_t size);
```

说明

函数 setvbuf 只能在 stream 指向的流和一个打开文件相关联之后，并且还没有对流执行任何其他操作之前使用。参数 mode 决定 stream 缓冲的方式，具体如下：_IOFBF 导致输入/输出完全缓冲；_IOLBF 导致输入/输出行缓冲；_IONBF 导致输入/输出不缓冲。如果 buf 不是一个空指针，可以使用它指向的数组来代替 setvbuf 函数分配的缓冲区[117]，size 指定了数组的大小，数组的内容在任何时候都是不确定的。

返回值

函数 setvbuf 成功时返回零，如果赋予参数 mode 一个无效的值或者请求没有被响应，就返回非零。

7.9.6 格式化的输入输出函数

fprintf

7.9.6.1 函数 fprintf

概述

```
#include <stdio.h>
int fprintf(FILE *stream, const char *format, ... );
```

说明

函数 fprintf 在 format 指向的串的控制下把输出写入 stream 指向的流，format 指向的串指定了输出时转换的后续参数的格式。如果格式的参数不够，那么这种行为未定义。如果格式用完了而参数有剩余，那么（通常）也处理多余的参数；否则，就忽略它们。当遇到格式串的结尾时，函数 fprintf 返回。

格式应该是一个多字节字符序列，以它的初始的转移状态开始和结束。格式由一条或者多条指示组成：普通的多字节字符（% 除外），它们原封不动地被复制到输出流；转换说明，每个转换说明都会产生零个或者多个后续的参数。它以 % 开头，% 后面依次出现以下元素：

- 零个或者多个标志字符（以任意顺序），修改转换说明的含义。
- 可选最小字段宽度。如果转换值的宽度比字段宽度小，那么就在字段的左边(如果给定了左调整标志，或者右边，下面会描述)填充空格（默认）。字段宽度是一个星号 *（后面会描述）的形式或者是一个十进制整数[118]。
- 可选的精度说明，它给出了 d、i、o、u、x 和 X 转换中出现的数字的最少个数；e、E 与 f 转换的小数点右边的位数；g 与 G 转换的最大有效位数；s 转换中要从字符串写入的最大字符数。精度说明的形式是小数点后面跟一个星号或者一个可

选的十进制整数。只有指定了句点的时候，采用的精度才是零。如果精度以其他转换说明符的形式出现，则这种行为未定义。

- 可选的字母 h 用于转换操作 d、i、o、u、x 或 X，表示转换参数的类型为 short int 或者 unsigned short int。（参数可能根据整值提升规则进行了提升，它的值在打印之前应该转换为 short int 或者 unsigned short int。）字符 h 用于 n 转换时，表示转换参数的类型是 short int*。一个可选的字母 l 用于转换操作 d、i、o、u、x 与 X 时，表示转换参数类型为 long int 或 unsigned long int；用于 n 转换时，表示转换参数的类型为 long int*。可选的字母 L 用于转换操作 e、E、f、或 g 与 G，表示转换参数的类型为 long double。如果 h、l 或者 L 跟其他的转换说明符一起出现，则这种行为未定义。
- 指定要使用的转换类型的字符。

像上面说明的那样，最小字段宽度或者精度，或者它们两个可以简化为一个星号。这种情况下，一个 int 参数提供了字段宽度或者精度。指定字段宽度或者精度或者它们两个的参数，应该顺序出现在所有要转换的参数之前。负的字段宽度参数表示为一个 "-" 标志后面跟一个正的字段宽度。负的精度参数被当作没有精度说明对待。

标志字符和它们的意思如下。

- 左对齐字段宽度中的值。（如果没有使用这个标志字符，则是右对齐。）

\+ 带符号转换的结果总是以一个 + 或 - 符号开始。（没有指定这个标志时，只有负值被转换时前面才带符号。

空格 如果一个带符号转换的结果中第一个字符不是符号或者转换的结果没有任何字符，就在结果前加一个空格。如果同时出现了空格和 + 标志，空格标志将被忽略。

\# 如果有 # 标志，结果被转换为一个"替换形式"。对 o 转换，它提高结果的精度来使结果的第一个数字为零。对 x（或者 X）转换，一个非零结果的前缀是 0x（或者 0X）。对 e、E、f、g 和 G 转换，即使小数点后面没有数，结果也要包含一个小数点。对 g 和 G 转换来说，尾部的零不从结果中删除。对其他的转换使用 # 标志，则行为是未定义的。

0 在 d、i、o、u、x、X、e、E、f、g 和 G 转换中，使用 0（后面跟符号或者基数指示符）作为填充字符。不执行空格填充。如果 0 标志和 - 标志同时出现，则忽略 0 标志。对 d、i、o、u、x 和 X 转换来说，如果指定了精度，则忽略 0 标志。对其他的转换使用 0 标志，则行为是未定义的。

转换说明符和它们的含义如下。

d、i int 参数转换为带符号的十进制数，形式为 *[-]dddd*。精度指定了显示数字的最小数目，如果转换的值可以用更少的数字表示，则前面用零填充。默认的精度是 1。把零值用精度零转换，结果没有任何字符。

o、u、x、X unsigned int 参数转换成无符号八进制数（o）、无符号十进制数（u）或者无符号十六进制数（x 或者 X），形式为 *dddd*。x 转换使用字母 abcdef，X 转换使用字母 ABCDEF。精度指定了显示数字的最小数目，如果转换的值可以用更少的数字表示，则前面用零填充。默认的精度是 1。把零值用精度零转换，结果没有任何字符。

f double 参数转换为十进制数，形式为 *[-]ddd.ddd*，这里小数点字符后面的位数和精度说明符相等。如果不说明精度，则默认为 6。如果精度为零并且没有使用 # 标志，则不会出现小数点字符。如果出现了一个小数点，则它的前面至少有一个数字。结果值为舍入到适当位数的数字。

e、E double 参数转换为十进制数，形式为 *[-]d.ddde±dd*，这里，小数点前面至少要有一个数字（如果参数非零，这个数字也非零），小数点后面的位数和精度

12

相等。如果不说明精度，则默认为 6。如果精度为零并且没有使用 # 标志，则不会出现小数点字符。结果值为舍入到适当位数的数字。E 转换说明符用 E 产生一个数来代替产生指数的 e。指数通常至少包含两位。如果值为零，指数就是零。

g、G double 参数转换为和 f 或 e（对 G 转换来说是 E）相同的样式，只不过精度指定的是有效数字的数目。如果精度是 0，就把它当作 1。使用的样式由转换的值决定，只有转换产生的结果的指数小于 -4 或者不小于精度的时候才使用 e（或者 E）样式。删除结果小数部分尾部的零，只有小数点后面有数字的时候才显示小数点。

c int 参数转换为 unsigned char 类型，并且写出结果。

s 参数应该是指向字符类型[119] 数组的指针。写出字符数组中结束的空字符之前的所有字符，如果指定了精度，则最多写出精度数目的字符。如果没有指定精度或者精度比数组的元素个数大，那么数组应该包含一个空字符。

p 参数应该是指向 void 的指针。参数的值按照实现定义的方式转换成一个可打印字符的序列。

n 参数是一个指向整型变量的指针，这个整型变量记录了到目前为止通过调用 fprintf 所写到输出流的字符的总数。不转换参数。

% 写出一个 %，不转换参数。完整的转换说明是 %%。

如果一个转换说明是无效的，则行为是未定义的[120]。

如果一个参数是，或者指向一个联合或者一个集合（除了使用 %s 转换的字符数组，或者使用 %p 转换的指针），则行为是未定义的。

任何情况下，一个不存在的字段宽度或者很小的字段宽度都不会导致字段的截短。如果转换结果比字段宽度长，那么就扩展字段来包含转换结果。

返回值

函数 fprintf 返回传送的字符的数目，如果发生了输出错误，就返回一个负值。

环境限制

一次转换产生的字符的最大数目最少应该是 509。

例子

下面的例子按照“Sunday，July 3，10:02”的形式打印日期和时间，后面是保留 5 位小数的 π 值

```
#include <math.h>
#include <stdio.h>
/*...*/
char *weekday, *month; /* pointers to strings */
int day, hour, min;
fprintf(stdout, "%s, %s %d, %.2d:%.2d\n",
      weekday, month, day, hour, min);
fprintf(stdout, "pi = %.5f\n", 4*atan(1.0));
```

fscanf

7.9.6.2 函数 fscanf

概述

```
#include <stdio.h>
int fscanf(FILE *stream, const char *format, ...);
```

说明

函数 fscanf 在 format 指向的串的控制下，使用后面的参数作为指向接收转换后的输入的对象的指针。从 stream 指向的流中读取输入。format 指向的串指定了可接收的输入序列和对它们进行转换以便赋值的方式。如果格式的参数不够，则行为是未定义的。如果格式用完了而参数有剩余，那么（通常）也处理多余的参数；否则，就忽略它们。

格式应该是一个多字节字符序列，以它的初始化的转移状态开始和结束。格式由零个或者多个指示性语句组成：一个或者多个空白字符，一个普通的多字节字符（不是 % 和空

白字符），或者一个转换说明。

每一个转换说明都由字符 % 引入，在 % 后面按顺序出现下面的元素。

- 可选的赋值取消字符 *。
- 可选的非零十进制整数，用来指定最大的字段宽度。
- 可选字符 h、l（ell）或者 L，用来说明接收对象的大小。如果一个参数是指向 short int 而不是 int 的指针，则 h 应该写在相应的转换说明符 d、i 和 n 之前；或者如果是指向 long int 的指针，则 l 写在之前。同理，如果参数是指向 unsigned short int 而不是 unsigned int 的指针，则 h 应该写在转换说明符 o、u 和 x 之前；或者如果是 unsigned long int，则 l 写在之前。最后，如果参数是指向 double 而不是 float 的指针，则 l 应该写在转换说明符 e、f 和 g 之前；或者如果是 long double，则 L 写在之前。如果 h、l 或者 L 和其他转换说明符出现在一起，则行为是未定义的。
- 一个字符，这个字符指定了待用的转换类型。下面会说明有效的转换说明符。

函数 fscanf 依次执行格式中的每条指示。如果一个指示失败了，就像下面描述的细节那样，函数 fscanf 就返回。失败是指输入失败（因为一个不可用的输入字符）或者匹配失败（因为不适当的输入）。

由任意个空字符组成的指示通过读取输入来执行，直到遇到第一个非空白字符（不对它进行读取），或者直到再也没有字符可以读取。

一个普通的多字节字符指示读取流的下一个字符。如果一个字符和组成控制语句的字符不同，则指示失败，不同的字符和它后面的字符都不能被读取。

一个转换说明符指示定义了匹配输入序列的一个集合，下面详细描述了每一个说明符。转换说明符按照以下的步骤执行：

跳过输入的空白字符（由 isspace 函数指定），除非说明包含一个 [、c 或者 n 说明符[121]。

从输入流中读取一个输入项，除非说明包含一个 n 说明符。一个输入项是输入字符的最长的匹配序列，除非它超过了指定的字段宽度，这种情况下它是序列中前面的该长度个字符。如果输入项后面还有字符的话，后面的第一个字符仍然没有被读取。如果输入项的长度为零，则指示执行失败：这种情况是匹配失败，除非一个错误阻止从流的输入，这种情况是输入失败。

除了一个 % 说明符的情况，输入项（或者在一个 %n 指示的情况下，输入字符的总数）被转换为和转换说明符适应的类型。如果输入项不是一个匹配的序列，指示的执行失败：这种情况是一个匹配失败。除非用一个 * 指示一个赋值取消，否则转换的结果存储在 format 后面的第一个参数指向的对象中，这个对象还没有接收到转换的结果。如果这个对象没有合适的类型，或者转换的结果不能在提供的空间中表示，则行为未定义。

下面的转换说明符是有效的。

d　和一个可选的有符号十进制整数相匹配，这个整数的格式和 strtol 函数 base 参数为 10 时的目标序列所期望的格式相同。相应的参数是一个指向整数的指针。

i　和一个可选的有符号整数相匹配，这个整数的格式与 strtol 函数 base 参数为 0 时的目标序列所期望的格式相同。相应的参数应该是一个指向整数的指针。

o　和一个可选的有符号八进制整数相匹配，这个整数的格式和 strtoul 函数 base 参数为 8 时的目标序列所期望的格式相同。相应的参数应该是一个指向无符号整数的指针。

u　和一个可选的有符号十进制整数相匹配，这个整数的格式和 strtoul 函数 base 参数为 10 时的目标序列所期望的格式相同，相应的参数应该是一个指向无符号整数的指针。

x　和一个可选的有符号十六进制数相匹配，这个整数的格式和 strtoul 函数 base 的参数为 16 时的目标序列所期望的格式相同。相应的参数是一个指向无符号整数的指针。

e、f、g 和一个可选的有符号浮点数匹配，该浮点数格式和函数 strtod 的目标序列所期望的格式相同。相应的参数是一个指向浮点数的指针。

s 和一个非空白字符[122]序列匹配。相应的参数应该是指向一个数组的首字符的指针，这个数组应足够大，可以接受该字符序列和一个结束的空字符，该字符会被自动添加。

[和一个非空字符[122]序列匹配，这些字符都属于一个字符集（扫描集）。相应的参数应该是指向数组的首字符的指针，这个数组应足够大，可以接受该序列和一个终止的空字符，该字符会被自动添加。转换说明符包括 format 串后面的所有字符，直到匹配的右括号（]），而且包括这个括号。两个括号之间的字符（扫描列表）组成了扫描集，除非左括号后面的字符是音调符号（^），这种情况下扫描集包含了音调符号和右括号之间所有不出现在扫描列表中的字符。如果转换说明符以 [] 或者 [^] 开头，那么右括号字符在扫描列表中，下一个右括号才是结束说明的匹配的右括号；否则，第一个右括号就是终止说明的右括号。如果“－”字符出现在扫描列表中，并且它不是第一个字符，也不是以 ^ 为第一个字符时的第二个字符，也不是最后一个字符，则行为是由实现定义的。

c 和一个字符序列匹配，字符的数目由字段宽度指定（如果指示中没有给出字段宽度，则数目为 1）。相应的参数为指向一个数组的首字符的指针，这个数组足够大，可以接受这个序列，不会添加空字符。

p 和一个实现定义的序列集合相匹配，这个序列集合应该和函数 fprintf 使用 %p 转换生成的序列集合相同。相应的参数是一个指向 void 的指针的指针。输入项的解释是由实现定义的。如果输入项是之前同一个程序执行过程中转换得到的一个值，那么产生的指针应该和这个值相等；否则，%p 转换的行为是未定义的。

n 不读取输入，相应的参数是指向一个整数的指针，这个整数记录了到目前为止通过调用 fscanf 函数从输入流读取的字符的总数。执行一个 %n 指示不会增加 fscanf 函数执行完成时返回的赋值总数。

% 和一个 % 匹配，不进行转换或者赋值。完整的转换说明应该是 %%。

如果转换说明是无效的，则行为未定义[123]。

转换说明符 E、G 和 X 也是有效的，它们的行为分别和 e、g 和 x 相同。

输入的时候如果遇到一个文件结束符，转换就会终止。如果在读入和当前指示匹配的任何字符之前出现了文件结束符（而不是允许出现开头的空白），则当前指示因输入失败而终止执行；否则，除非当前指示以一个匹配失败终止执行，下一条指示（如果有的话）就因输入失败终止执行。

如果转换因为一个无法识别的输入字符而终止，那么这个输入字符留在输入流中不被读取。尾部的空白（包括换行符）也留在输入流中不被读取，除非它和一个指示匹配。和通过使用 %n 指示相比，文字匹配和取消赋值的赋值取消标志的成功执行不能被直接确定。

返回值

如果在任何转换之前发生了输入失败，函数 fscanf 返回宏 EOF 的值；否则，函数 fscanf 返回赋值的输入项的数目，这个值会比早期发生了匹配失败时提供的值小，甚至有可能为 0。

例子

以下代码段

```
#include <stdio.h>
/*. . .*/
int n, i; float x; char name [50];
n = fscanf(stdin, "%d%f%s", &i, &X, name);
```

的输入

```
25 54.32E-1 thompson
```

会使 *n* 的值为 3，*i* 的值为 25，*x* 的值为 5.432，*name* 包括 thompson\0。

以下程序段

```
#include <stdio.h>
/*...*/
int i; float x; char name[50];
fscanf(stdin, "%2d%f%*d %[0123456789]", &i, &x, name);
```

的输入

```
56789 0123 56a72
```

会使 *i* 的值为 56，*x* 的值为 789.0，会跳过 0123，*name* 将包括 56\0。从输入流读取的下一个字符将会是 a。

下面的代码可以重复地从 stdin 中接受一个数量、一个测量单元和一个项目名字：

```
#include <stdio.h>
/*...*/
int count; float quant; char units[21], item[21];
while (!feof(stdin) && !ferror(stdin)) {
      count = fscanf(stdin, "%f%20s of %20s",
            &quant , units, item);
      fscanf(stdin," %*[^\n]");
}
```

如果 stdin 流包含下面各行：

```
2 quarts of oil
-12.8degrees Celsius
lots 0f luck
10.0LBS     of
dirt
100ergs of energy
```

则上面例子的执行就和下面的赋值语句类似：

```
quant = 2; strcpy(units, "quarts"); strcpy(item, "oil");
count = 3;
quant = -12.8; strcpy(units, "degrees");
count = 2: /* "C" fails to match "o" */
count = 0; /* "1" fails to match "%f" */
quant = 10.0; strcpy (units, "LBS") ; strcpy (item, "dirt");
count = 3;
count = 0; /* "100e" fails to match "%f" */
count = EOF;
```

参见：函数 strtod（7.10.1.4）、函数 strtol（7.10.1.5）和函数 strtoul（7.10.1.6）。

printf

7.9.6.3 函数 printf

概述

```
#include <stdio.h>
int printf(const char *format, ...);
```

说明

函数 printf 和 fprintf 等价，只不过 fprintf 把参数 stdout 插到了 printf 的参数的前面。

返回值

函数 printf 返回传送的字符数，发生输出错误的时候，就返回一个负值。

scanf

7.9.6.4 函数 scanf

概述

```
#include <stdio.h>
int scanf(const char *format, ...);
```

说明

函数 scanf 和 fscanf 等价，只不过 fscanf 把参数 stdin 插到 scanf 的参数之前。

返回值

如果在任何转换之前发生了输入失败，函数 scanf 返回宏 EOF 的值；否则，函数 scanf 返回赋值的输入项的数目，这个值可能比早期发生了匹配失败的情况提供的值小，或者甚至是零。

sprintf

7.9.6.5 函数 sprintf

概述

```
#include <stdio.h>
int sprintf(char *s, const char *format, ...);
```

说明

函数 sprintf 等价于 fprintf，只不过参数 s 指定了一个数组，生成的输出写入到这个数组中，而不是写到一个流中。在写入的字符最后写入一个空字符，它不计入返回和的一部分。如果两个交迭的对象之间发生了复制，则行为未定义。

返回值

函数 sprintf 返回写到数组中的字符的数目，不包括终止的空字符。

sscanf

7.9.6.6 函数 sscanf

概述

```
#include <stdio.h>
int sscanf(const char *s, const char *format, ...);
```

说明

函数 sscanf 和 fscanf 等价，只不过参数 s 指定了一个串，输入是从这个串中获取，而不是从流中获取。到达串的结尾等价于函数 fscanf 遇到了文件结束符。如果两个交迭的对象之间发生了复制，则行为未定义。

返回值

如果在任何转换之前发生了输入错误，函数 sscanf 返回宏 EOF 的值。否则，函数 sscanf 返回输入项赋值的数目，这个值可能比早期发生了匹配错误的情况下提供的值小，甚至可能为零。

vfprintf

7.9.6.7 函数 vfprintf

概述

```
#include <stdarg.h>
#include <stdio.h>
int vfprintf(FILE *stream, const char *format, va_list arg);
```

说明

函数 vfprintf 等价于函数 fprintf，只不过可变参数表用 arg 代替，它已经被宏 va_start（可能后来还有 va_arg 调用）初始化了。函数 vfprintf 不调用宏 va_end[124]。

返回值

函数 vfprintf 返回传送的字符的数目，如果发生了输出错误，则返回一个负值。

例子

下面的例子展示了常规的错误报告中对 vfprintf 的使用

```
#include<stdarg.h>
#include<stdio.h>

void error(char *function_name, char *format, ... )
{
      va_list args;
```

```
        va_start (args, format) ;
        /* print out name of function causing error */
        fprintf(stderr, "ERROR in %s: ", function_name);
        /* print out remainder of message */
        vfprintf (stderr, format, args) ;
        va_end(args) ;
    }
```

vprintf

7.9.6.8 函数 vprintf

概述

```
#include <stdarg.h>
#include <stdio.h>
int vprintf(const char *format, va_list arg);
```

说明

函数 vprintf 等价于 printf，只不过用 arg 代替可变参数表，它已经被宏 va_start（可能后来还有 va_arg 调用）初始化了。函数 vprintf 不调用宏 va_end[124]。

返回值

函数 vprintf 返回传送的字符的数目，发生了输出错误的时候，返回一个负值。

vsprintf

7.9.6.9 函数 vsprintf

概述

```
#include <stdarg.h>
#include <stdio.h>
int vsprintf(char *s, const char *format, va_list arg);
```

说明

函数 vsprintf 等价于 sprintf，只不过可变参数表用 arg 代替，它已经被宏 va_start（可能后来还有 va_arg 调用）初始化了。函数 vsprintf 不调用宏 va_end[124]。如果两个交迭的对象之间发生了复制，则行为未定义。

返回值

函数 vsprintf 返回写入数组中的字符的数目，不包括终止的空字符。

7.9.7 字符输入/输出函数

fgetc

7.9.7.1 函数 fgetc

概述

```
#include <stdio.h>
int fgetc(FILE *stream);
```

说明

函数 fgetc 从 stream 指向的输入流中读取下一个字符（如果有的话），并把它由 *unsigned char* 类型转换为 int 类型，并且流的相关的文件定位符（如果定义的话）向前移动一位。

返回值

函数 fgetc 返回 stream 指向的输入流中的下一个字符。如果输入流在文件结束处，则设置文件结束符，函数 fgetc 返回 EOF。如果发生了读错误，则设置流的错误指示符，函数 fgetc 返回 EOF[125]。

fgets

7.9.7.2 函数 fgets

概述

```
#include <stdio.h>
char *fgets(char *s, int n, FILE *stream);
```

说明

函数 fgets 从 stream 指向的流中读取字符，读取字符的数目最多比 n 指定的数目少 1，然后将字符写入到 s 指向的数组中。读入换行符（保留）或者文件结束之后，不再读

12

取其他的字符。最后一个字符写入数组后立即写入一个空字符。

返回值

如果成功执行，则函数 fgets 返回 s。如果遇到文件结束并且没有向数组中写入字符，则数组的内容不变且返回一个空指针。如果操作的过程中发生了读错误，则数组的内容不确定，并返回一个空指针。

fputc

7.9.7.3 函数 fputc

概述

```
#include <stdio.h>
int fputc(int c, FILE *stream);
```

说明

函数 fputc 把 c 指定的字符（转换为 unsigned char 类型）写到 stream 指向的输出流中相关的文件定位符（如果定义的话）指定的位置处，并把文件定位符向前移动适当的位置。如果文件不支持定位要求或者流以附加模式打开，字符就添加到输出流的后面。

返回值

函数 fputc 返回写入的字符。如果发生了写错误，则设置流的错误指示符，函数 fputc 返回 EOF。

fputs

7.9.7.4 函数 fputs

概述

```
#include <stdio.h>
int fputs(const char *s, FILE *stream);
```

说明

函数 fputs 把 s 指向的串写入 stream 指向的流中，不写入结束的空字符。

返回值

如果发生了写错误，则函数 fputs 返回 EOF。否则，它返回一个非负值。

getc

7.9.7.5 函数 getc

概述

```
#include <stdio.h>
int getc(FILE *stream);
```

说明

函数 getc 等价于函数 fgetc，只不过如果 getc 作为一个宏实现，它可能会对 stream 进行多次计算，所以它的参数不能是具有副作用的表达式。

返回值

函数 getc 返回 stream 指向的输入流的下一个字符。如果流处于文件结束处，则设置该流的文件结束指示符，函数 getc 返回 EOF。如果发生了读错误，则设置流的错误指示符，函数 getc 返回 EOF。

getchar

7.9.7.6 函数 getchar

概述

```
#include <stdio.h>
int getchar(void);
```

说明

函数 getchar 等价于用参数 stdin 调用的函数 getc。

返回值

函数getchar返回stdin指向的输入流的下一个字符。如果流处于文件结束处，则设置流的文件结束符，函数getchar返回EOF。如果发生了读错误，则设置流的错误指示符，getchar返回EOF。

gets

7.9.7.7 函数gets

概述

```
#include <stdio.h>
char *gets(char *s);
```

说明

函数gets从stdin指向的输入流中读取若干个字符，并将其保存到s指向的数组中，直到遇到文件结束或者读取一个换行符。所有的换行符都被丢弃，在最后一个字符读到数组中之后立即写入一个空字符。

返回值

函数gets成功执行时返回s。如果遇到了文件结束并且数组中没有读入任何字符，则数组的内容保持不变，返回一个空指针。如果操作的过程中发生了读错误，则数组的内容不确定，返回一个空指针。

putc

7.9.7.8 函数putc

概述

```
#include <stdio.h>
int putc(int c, FILE *stream);
```

说明

函数putc等价于fputc，只不过如果putc作为一个宏实现，它可能会对stream计算多次，所以它的参数不能是有副作用的表达式。

返回值

函数putc返回写入的字符。如果发生了写错误，则设置流的错误指示符，putc返回EOF。

putchar

7.9.7.9 函数putchar

概述

```
#include <stdio.h>
int putchar(int c);
```

说明

函数putchar等价于把stdout作为第二个参数调用的putc。

返回值

函数putchar返回写入的字符。如果发生了写错误，则设置流的错误指示符，putchar返回EOF。

puts

7.9.7.10 函数puts

概述

```
#include <stdio.h>
int puts(const char *s);
```

说明

函数puts把s指向的串写到stdout指向的流中，并且在输出最后添加一个换行符。不写入结束的空字符。

返回值

如果发生了写错误，函数puts返回EOF。否则，它返回一个非负值。

ungetc

7.9.7.11　函数 ungetc

概述

```
#include <stdio.h>
int ungetc(int c, FILE *stream);
```

说明

函数 ungetc 把 c 指定的字符（转换为 unsigned char 类型）退回到 stream 指向的输入流中。退回字符是通过对流的后续读取并按照退回的反顺序来返回的。如果中间成功调用（对同一个流）了一个文件定位函数（fseek、fsetpos 或者 rewind），那么就会丢弃流的所有退回的字符。流的相应的外部存储保持不变。

退回的一个字符是受到保护的。如果对同一个流调用 ungetc 函数太多次，中间又没有读或者文件定位操作，那么这种操作可能会失败。

如果 c 的值和宏 EOF 的值相等，则操作失败且输入流保持不变。

对 ungetc 的成功调用会清空流的文件结束符。读取或者丢弃所有回退的字符之后，流的文件定位符的值和这些字符回退之前的值相同。对文本流来说，在对函数 ungetc 的一次成功调用之后，它的文件定位符的值是不明确的，直到所有回退字符被读取或者丢弃为止。对二进制流来说，每成功调用一次 ungetc 之后它的文件定位符都会减 1。如果在一次调用之前它的值是零，那么调用之后它的值是不确定的。

返回值

函数 ungetc 返回转换后的回退字符。如果操作失败，则返回 EOF。

参见：文件定位函数（7.9.9）。

7.9.8　直接输入 / 输出函数

fread

7.9.8.1　函数 fread

概述

```
#include <stdio.h>
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
```

说明

函数 fread 从 stream 指向的流中读取最多 nmemb 个元素到 ptr 指向的数组中，nmemb 的大小是由 size 指定的。流的文件定位符（如果定义的话）根据成功读取的字符数向前移动。如果发生了错误，流的文件定位符最后的值是不确定的。如果只读取了部分元素，它的值也是不确定的。

返回值

函数 fread 返回成功读取的元素的数目，如果发生了读错误或者遇到了文件结束，返回值可能比 nmemb 小。如果 size 或者 nmemb 为零，则 fread 返回零且数组的内容和流的状态保持不变。

fwrite

7.9.8.2　函数 fwrite

概述

```
#include <stdio.h>
size_t fwrite(const void *ptr, size_t size, size_t nmemb,
      FILE *stream);
```

说明

函数 fwrite 从 ptr 指向的数组中读取最多 nmemb 个元素并将其写到 stream 指向的流中，nmemb 的大小是由 size 指定的。流的文件定位符（如果定义的话）根据成功写入的字符数向前移动。如果发生了错误，流的文件定位符最后的值是不确定的。

返回值

函数 fwrite 返回成功写入的元素的数目。如果遇到了写错误，它可能比 nmemb 小。

7.9.9 文件定位函数

fgetpos

7.9.9.1 函数 fgetpos

概述

```
#include <stdio.h>
int fgetpos(FILE *stream, fpos_t *pos);
```

说明

函数 fgetpos 把 stream 指向的流的文件定位符的当前值存储到 pos 指向的对象中。存储的值包含了未指明的信息，函数 fsetpos 可以使用这些信息来使流重新定位到这次调用函数 fgetpos 时的位置。

返回值

如果操作成功，函数 fgetpos 返回零；如果不成功，函数 fgetpos 返回非零并且把一个由实现定义的正值存储在 errno 中。

参见：函数 fsetpos（7.9.9.3）。

fseek

7.9.9.2 函数 fseek

概述

```
#include <stdio.h>
int fseek(FILE *stream, long int offset, int whence);
```

说明

函数 fseek 为 stream 指向的流设置文件定位符。

对二进制流来说，新的位置从文件的开始以字符为单位进行测量，通过在 whence 指定的位置上加上 offset 来获得。如果 whence 是 SEEK_SET，指定的位置就是文件的开始位置，如果是 SEEK_CUR，就是文件的当前位置，如果是 SEEK_END，则是文件的结束位置。二进制流不需要严格支持 whence 为 SEEK_END 时的 fseek 调用。

对文本流来说，offset 或者是零，或者是前面对同一个流调用 ftell 函数返回的值，whence 应该是 SEEK_SET。

对函数 fseek 的成功调用会清空流的文件结束符并且解除 ungetc 函数对这个流的任何影响。调用 fseek 之后，对更新后的流的下一个操作可能是输入或者输出。

返回值

只有当不能满足请求时，函数 fseek 才返回非零值。

参见：函数 ftell（7.9.9.4）。

fsetpos

7.9.9.3 函数 fsetpos

概述

```
#include <stdio.h>
int fsetpos(FILE *stream, const fpos_t *pos);
```

说明

函数 fsetpos 根据 pos 指向的对象的值来设置 stream 指向的流的文件定位符，pos 指向的对象的值应该是前面对同一个流调用函数 fgetpos 得到的返回值。

对函数 fsetpos 的成功调用会清空流的文件结束符并且解除 ungetc 对这个流的影响。调用 fsetpos 之后，对更新后的流的下一个操作可能是输入或输出。

返回值

如果操作成功，函数 fsetpos 返回零；如果不成功，函数 fsetpos 返回非零并且把一个由实现定义的正值存储在 errno 中。

ftell

7.9.9.4 函数 ftell

概述

```
#include <stdio.h>
long int ftell(FILE *stream);
```

说明

函数 ftell 获得 stream 指向的流的文件定位符的当前值。对一个二进制流来说，这个值是从文件开头到当前位置的字符的数目；对一个文本流来说，它的文件定位符包含了未说明的信息，函数 fseek 可以使用这些信息来使流的文件定位符回到这次调用 ftell 时的位置。这两个返回值的差值不一定可以作为读取或者写入的字符数使用。

返回值

如果成功执行，函数 ftell 返回流的文件定位符的当前值。如果失败了，函数 ftell 返回 -1L 并且把一个实现定义的正值存储在 errno 中。

rewind

7.9.9.5 函数 rewind

概述

```
#include <stdio.h>
void rewind(FILE *stream);
```

说明

函数 rewind 把 stream 指向的流的文件定位符设置在文件的开始位置，它等价于

```
(void)fseek(stream, 0L, SEEK_SET)
```

只不过流的错误指示符也被清零。

返回值

函数 rewind 没有返回值。

7.9.10 错误处理函数

clearerr

7.9.10.1 函数 clearerr

概述

```
#include <stdio.h>
void clearerr(FILE *stream);
```

说明

函数 clearerr 清空 stream 指向的流的文件结束符和错误指示符。

返回值

函数 clearerr 没有返回值。

feof

7.9.10.2 函数 feof

概述

```
#include <stdio.h>
int feof(FILE *stream);
```

说明

函数 feof 测试 stream 指向的流的文件结束符。

返回值

当且仅当 stream 流设置了文件结束符时函数 feof 返回一个非零值。

ferror

7.9.10.3 函数 ferror

概述

```
#include <stdio.h>
int ferror (FILE *stream);
```

说明

函数 ferror 测试 stream 指向的流的错误指示符。

返回值

当且仅当 stream 流设置了错误指示符时函数 ferror 返回非零。

perror

7.9.10.4 函数 perror

概述

```
#include <stdio.h>
void perror(const char *s);
```

说明

函数 perror 把整数表达式 errno 中的错误编号转换为一条错误信息。它把一个字符序列写到一个这样的标准错误流：首先（如果 s 不是空指针并且 s 指向的字符不是空字符），s 指向的串后面跟一个冒号（:）和一个空格；然后是一个适当的错误信息串后面跟一个换行符。错误信息串的内容和用参数 errno 调用函数 strerror 的返回值相同，这是由实现定义的。

返回值

函数 perror 没有返回值。

参见：函数 strerror（7.11.6.2）。

脚注

110. 如果实现没有强制限制文件名串的长度，FILENAME_MAX 的值应该是可以容纳一个文件名串的数组被推荐的大小。当然，文件名串的内容又受限于其他系统指定的约束，因此并不是所有长度为 FILENAME_MAX 的串都可以成功打开。
111. 实现不需要区分文本流和二进制流。在这样的实现下，文本流中不需要使用换行符，每一行的长度也没有限制。
112. 在 Base Document 中它被描述为文件指针，这个国际标准没有使用这个术语是为了避免和指向 FILE 类型的对象的指针发生混淆。
113. 实现可能导致函数 rename 失败的原因是文件已经打开或者有必要复制它的内容来使重命名生效。
114. 使用 tmpnam 函数生成的串创建的文件是临时的，只是说它们的名字不应该和那些通过实现的常规命名规则生成的名字冲突。当对这些文件的使用结束时，仍然有必要使用函数 remove 在程序终止之前来删除它们。
115. 这些序列后面可能跟一些附加字符。
116. 函数 freopen 的基本作用是改变和一个标准文本流（stderr、stdin 或者 stdout）相关联的文件，因为那些标识符不必是可更改的左值，函数 fopen 的返回值可以赋给它。
117. 缓冲的生存期至少要和打开的流一样长，所以当缓冲为自动存储类型时，打开的流应该在缓冲释放之前关闭，
118. 注意 0 只是一个标志，而不是字段宽度的开始。
119. 对多字节字符没有特别的规定。
120. 参考“库的展望”（7.13.6）。
121. 这些空白字符不计入指定的字段宽度。
122. 对多字节字符没有特别的规定。
123. 参考“库的展望”（7.13.6）。
124. 当函数 vfprintf、vsprintf 和 vprintf 调用宏 va_arg 时，返回后 arg 的值是不确定的。
125. 文件结束和读错误可以通过使用函数 feof 和 ferror 来区分。

12.3 <stdio.h> 的使用

<stdio.h> 中声明的大多数函数是对一个流进行操作，这个流和一个打开的文件相关联。一旦执行了程序，就可以使用下面 3 个流：

stdin

stdout

stderr

- stdin——读取文本的标准源（标准输入流）。
- stdout——写入文本的标准目的地（标准输出流）。
- stderr——输出错误信息的标准目的地（标准错误流）。

<stdio.h> 中声明的许多函数都用到了这些流，用户不需要对它们命名。对那些需要一个流参数的函数，可以选其中一个名字作为流参数。

打开文件

也可以通过名字打开一个文件，并和一个流连接起来。可以通过调用函数 fopen 或 freopen 把一个流和一个打开的文件关联起来，例如：

```
fptr = fopen (fname, fmode);
fptr = freopen(fname, fmode, fptr);
```

只有当这两个函数使用模式 fmode 打开文件名为 fname 的文件，并且把该文件和 fptr 指向的对象控制的流联系起来时，它们才返回指向 FILE 类型的指针类型的非空值。

fptr 只能作为 C 标准库中其他流的 I/O 服务函数的参数使用。不要试图探索它所指向的数据对象的内部，即使某个实现在 <stdio.h> 中提供了可以显示某些域的 FILE 声明。不要试图修改任何一个域。更不要试图将其内容复制到另一个类型为 FILE 的数据对象中并用此副本来代替它，因为实现可以认为它们知道控制流的数据对象的所有有效地址。（换句话说，函数 fopen 返回的地址可能还有其他作用，不仅仅是存储在该地址上的值。）

一旦通过成功调用函数 fclose（或是局部成功调用函数 freopen）关闭了一个流，就不要再使用它对应的 fptr 的值。它所指向的存储空间将会被重新分配或是重新利用。（更不要复制指针的值。严格来说，实现发现一个指针指向了一个已经被重新分配的空间时就会崩溃。）

FILE 类型

用户也不需要知道 FILE 类型数据对象的内部表示，他们只需要知道在其他东西中可以通过某种方式表示它：

- 文件结束符记录是否需要结束此文件。
- 错误指示符记录读或写是否导致不可恢复的数据传输错误。
- 文件定位符记录从文件中读出或写入的下一个字节（这在某些类型的文件中可能没有定义）。
- 缓冲信息记录任何读写缓冲区是否存在和它的大小。
- 状态信息决定下一个是读操作还是写操作。

对于正在命名的文件，最好避免把任何文件名写入代码中。（这是一个

好办法，其中的原因有很多。）如果要输入或创建一个文件名，使用一个可以容纳 FILENAME_MAX（<stdio.h> 中定义了这个宏）个字符的缓冲区。假设一个文件名是一个常规的以空字符结尾的串。不要关心内部实现，也不要删除文件名中的任何字符。

如果必须创建文件名，例如头文件名，原则是越简单越好。任何实现几乎都可以接受由 1 到 6 个英文字母，后面跟一个点，后面再跟一个英文字母构成的文件名。例如 "myhdr.h" 和 "X.Y"。不要认为这些字符区分大小写，也可能不区分。不要期望这些名字在操作系统中不修改就可以使用。C 标准库可能需要将它们映射为某些其他的形式来遵守当地用法。

模式

文件模式是由以下 3 个字母中的一个开头的字符串。

- r 说明要打开已存的文件用来读取数据。
- w 说明要打开已存的文件用来写入数据并且丢弃原来的数据，或者创建一个初始时没有任何内容的新文件。
- a 与 w 的作用一样，只是增加了一条：在对流进行写操作之前，将文件定位符置于文件结束处。
- 模式后面也可以加上以下两个可选的字符，顺序不定。
- + 说明可以对打开用来读取数据（用 r）的文件进行写操作，或者可以对打开用来写数据（用 w 或 a）的文件进行读操作。
- b 说明打开一个二进制文件而不是文本文件。

还可以在上述字符之后加上其他的字符。每个实现都可以定义哪些附加的参数（如果有的话）可以作为 fmode 的——部分来编写。例如，系统允许编写下面的代码：

```
fopen (fname, "w, lrecl=132, recfm=fixed")
```

在 System/370 中，至少有一种 C 实现认为上述语句是请求新建一个定长记录即每个记录长度为 132 字节的文件。然而，没有任何标准规定定义的模式后面的内容。如果程序要在各种实现之间移植，使用其他模式的 fopen 调用可能会失败或者出现隐秘的错误。

读和写

C 标准库为流的读和写提供了大量的函数，例如，可以读取单个字符，读取一定长度的字符，或是读取多个字符并将它们转化为格式串控制的某种编码形式。

函数 fgetc

这里详细地定义了函数 fgetc 读取单个字符的过程，其他所有函数都被定义为好像多次调用函数 fgetc 来获得输入字符，不管它们实际是否这样做。函数 fgetc 首先保证流支持普通的读操作同时一个读请求被及时接受（参考本节后面的“缓冲区控制”部分）；然后决定是否需要给流分配一个缓冲区，如果需要，则请求分配；接着它确定是否需要执行一个物理读操作

（填满一个空的缓冲区或者直接输入字符），如果需要，就执行这个操作。它在发生物理读操作错误时会设置一个错误指示符，或者在其文件结束处发生物理读操作时设置文件结束符。完成以上操作后就有一个字符需要传送，该函数就传送它并且将文件定位符向前移动一个字符。

对每个字符都执行所有这些操作的实现一定很慢。所以就不必奇怪为什么在过去几年里实现者尽最大的努力删减那些不必要的操作。一个主要的技巧是尽可能少地进行物理读操作，一次操作尽可能多地读取字符，然后简要地记录数据流的状态以快速测试每个字符。事实上，函数 getc 传统上是一个宏，它是函数 fgetc 的加速版本。

不安全的宏

标准 C 要求函数 getc 也要作为一个真正的函数来表示。头文件 <stdio.h> 能够，也通常用宏来屏蔽这个函数声明。这个宏能够且通常一定会纵容对指向 FILE 参数的指针进行多次值计算的不安全操作。头文件也能用一个宏定义来屏蔽函数 fgetc(或是其他任何函数)。唯一的不同是一定要保证宏而不是函数 getc（和 putc），对它的每一个参数只执行一次求值操作，这样求值的副作用和调用一个真正的函数的相同。

函数 fputc

写操作与读操作很相似。原始函数 fputc 可将一个字符写入流。函数 fputc 首先确保数据流支持普通的写操作且一个写请求及时被响应（参考本节后面的“缓冲区控制”部分）；然后决定是否需要给流分配一个缓冲区，如果需要，就请求分配；接着决定是否需要执行一个物理写操作（释放一个满的缓冲区或是直接输出字符），如果需要，就执行这项操作。它在发生物理写错误时设置错误指示符，完成以上工作后，如果字符被送进来，该函数就让文件定位符向前移动一个字符。同样，一个典型的实现会用一个可能不安全的屏蔽宏定义来实现相关函数 putc。

文件定位

从开始到结尾连续写或读一个流是很普遍的。事实上，许多来自终端和管道的像流一样的伪文件只能以这种方式处理。不过，当需要重新处理数据或是以随机顺序处理数据时就会出现例外。这些例外情况需要用各种方法改变文件定位符的正常行进。C 标准库为此提供了 3 种不同的机制来修改文件定位符：

- ungetc 使刚从一个流中读出的字符退回流中。
- fseek、ftell 和 rewind 记住文件定位符并将其恢复到之前的位置，假定文件定位符可以编码为 *long* 类型。
- fgetpos、fsetpos 和 rewind 记住一个任意的文件定位符并将其恢复到以前的位置。

函数 ungetc 函数 ungetc 甚至能在不支持文件定位请求的流上工作，例如来自终端或是管道的流。它可以回退一个和刚才读取的不同字符。如果在对一个流的第一次读取之前调用这个函数，它甚至可以在文件开始之前回退一个字符。

然而，在两个读操作之间，各种实现允许回退字符的数目也不同。即使不连续地调用格式化输入函数（例如 scanf），且这些函数也需要一个回退字符，这时也可以保证一个回退字符。对于一个可移植的程序，不要认为可以回退多个字符。

函数 ungetc 与定位文件的其他两个机制之间的联系很小。X3J11 委员会花了很多时间去整理调用函数 ungetc 和 fseek 的各种序列的语义。一般规则是通过函数 ungetc 回退的字符会在任何其他文件定位请求之后消失。但是仔细阅读函数说明，就能确保得到期望的结果。我的建议是除了读请求外不要将函数 ungetc 的调用与其他函数调用混合使用。

fseek
ftell
rewind

函数 fseek 和 ftell（和 rewind）是 C 早期发展的传统的文件定位函数，它们假定文件定位符可以编码为 *long* 类型，就像 12.1 节中所提到的。这个假设在 UNIX 中成立，因为在 UNIX 中文件的长度不会超过 2^{32} 个字节而且可以将文件定位在任意字节上。但是对于支持更大文件或是需要更详细的文件定位信息的系统，这个假设可能不成立。

例如，一个文本文件可能被结构化为块和块内记录——将一个块编号，记录编号和记录内部的偏移量打包成 *long* 类型可能会需要不可能实现的任意一个字节来协调。由于这些原因，函数 ftell 可能会失败（返回 -1），而不是返回一个文件定位符的错误编码。

可以使用 fseek 和 ftell 提前随机访问一个二进制文件（如果文件不是太大）的若干个字节。在这种情况下，编码的文件定位符就是从当前位置到文件开头（即字节为零的位置）的以字节为单位的偏移量。可以用这样的文件定位符执行算术运算，或是从头开始计算它们，而且肯定能得到期望的字节数。

但是文本文件中编码的文件定位符在不同实现中有不同的格式。使用 ftell 将会得到一个标志文件当前位置的信息。（如果它不能对当前文件定位符进行编码，它就会返回一个失败编码。）在同一程序后面的执行过程中且关闭这个文件之前，传递给 fseek 相同的值可以使文件定位符恢复较早的值。不要认为可以将这样的值从一个程序的执行保存到下一个程序的执行中，或者是从一个打开的文件保存到下一个打开的文件中，因为实现对这种编码采用的方式可能会非常奇怪。

12

fgetpos
fsetpos

如果只需要重新定位之前访问过的文件位置，那么就应该使用第三种机制。委员会添加了函数 fgetpos 和 fsetpos 来支持任意大小和结构的文件的定位。这些函数工作时要使用 <stdio.h> 中定义的 fpos_t 类型的值，而该类型的结构和一个实现对任意文件定位符进行编码的结构一样复杂。假设 fpos_t 是一个只能对它进行复制、作为函数参数传递或者作为一个函数值来接收的结构类型。甚至对于一个二进制文件，也没有定义一种方式来对这些值进行比较或是执行算术运算。

缓冲区控制

原则上，用户可以在一定程度上对 I/O 函数操作流时缓冲数据的方式进行控制，但是缓冲是以各种 I/O 模式的推测为基础的一个优化，这些推测通常都是正确的，并且很多实现都遵循用户的建议，但是它们并不是一定要这么做。实现可以随意忽略用户大部分的缓冲请求。

setvbuf
setbuf

然而，如果一个更大的缓冲区可以提高性能或者一个更小的缓冲区可以节省空间，用户还可以提供自己的后备缓冲区。在打开文件之后且对流进行任何其他操作之前，调用函数 setvbuf。（避免使用旧函数 setbuf，它的灵活性较差。）可以指定 I/O 是完全缓冲、行缓冲，还是无缓冲。这可能会影响程序的性能。

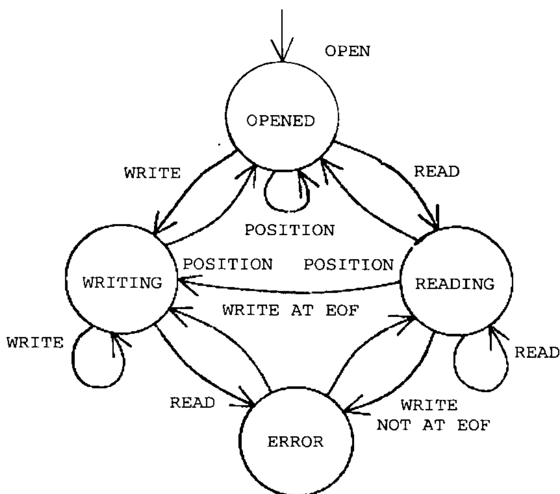
函数 fflush

有时可能需要大部分时间内都进行缓冲，但又要对输出流被清空到外部设备的时间进行有限的控制。函数 fflush 就可以保证被调用的时候，有一个或更多的流清空到外部。这有利于在一个交互式环境中输出信息，它也可以使数据库更加健壮以便防止偶然出现的程序崩溃。然而在标准 C 中函数 fflush 对输入流的影响没有任何定义。不能像在 UNIX 下那样调用这个函数在提示符出现之前可靠地丢弃输入。

C 标准库不允许某些模式的读和写。基本规则是在中间没有文件定位请求的情况下，一个读操作后不能紧跟着一个写操作，或是一个写操作不能紧跟着一个读操作。再具体一点，中间调用必须是函数 fflush、fseek、fsetpos 或 rewind。设置了文件结束指示器的读操作后面可紧跟一个写操作，但是，非常奇怪的是，前面有一个隐式的寻找（对一个以 fmode 模式打开的以 a 开头的文件）的写操作后面不可以紧跟一个读操作。图 12-1 是一个总结这些规则的状态转移图。

最后一条建议是尽可能地给 I/O 流函数自由。不要试图严格地控制缓冲。可以让一个实现对它们进行优化，而所有其他实现不对它们进行优化。而且不要任意地将读操作、写操作和各种文件定位操作混合使用，因为这样可能会使实现崩溃，也很容易使你自己的程序崩溃。

图 12-1
流的状态



格式化输出

输入 / 输出的一个重要方面是执行格式化输出。I/O 几乎总是在 C 语言中最早接触的，就像下面这个流行的第一个程序：

```
#include <stdio.h>

int main(void)
    { /* say hello */
    printf("hello world\n");
    }
```

除非只编写嵌入式程序，否则格式化输出很可能是用户必须掌握的一种最重要的 I/O 方法。

一个程序可以产生只有另一个计算机程序喜欢或者理解的输出。如果两个程序在同一个体系结构中运行，那么它们共用对数据编码的符号。一个程序可以编写整型和浮点型的标量，甚至是结构和联合，而且另一个程序可以立即读取它们并对它们进行操作。程序间可以通过从二进制文件复制字节或是将字节复制到该文件中来共享除指针类型以外的其他任何类型的数据。

然而，如果想在运行于不同计算机体系结构下的程序间共享数据，就一定要更加小心。不同的计算机对整数和浮点数编码的方式通常是不同的。即使两个计算机在数量级和编码方式上相同，它们在内存中存储一个多字节数据对象的各个字节的顺序也通常不相同。

不同计算机对存储对齐的要求也有很大区别，所以结构内部（和结构与联合的尾部）的空隙变化比预期的要多。除非你非常细心，否则最好不要把二进制文件作为数据交换的媒介。

文本文件有以下 3 个重要的方面优于二进制文件。

- 只有人才能生成和修改它们。
- 它们可以被写到一台打印机或是一个终端上，而且人们基本上可以理解其显示的内容。
- 它们可以被不共享数据编码方式的程序共享。

打印函数

为编码的数据设计一种文本表示方法的过程叫做*输出格式化*。打印函数（都将 print 作为它们名字的一部分且都在 <stdio.h> 中声明）产生格式化输出。要使用这些打印函数，必须知道如何调用它们，如何说明格式，以及它们将会执行什么转换。C 标准库提供了 6 个不同的打印函数，声明如下：

```
int fprintf (FILE *stream, const char *format, ... );
int printf (const char *format, ...);
int sprintf (char *dest, const char *format, ... );
int vfprintf (FILE * stream, const char *format, va_list ap);
int vprintf (const char *format, va_list ap);
int vsprintf (char *dest, const char *format, va_list ap);
```

所有的函数都接受一个 format 参数，它是一个指向只读的以空字符结尾的字符串的指针。这个格式告诉函数期望附加的参数是什么（如果有的话）以及如何转换它们。它也指定了散布在任意转换后的参数之间的字面量文本。下面会详细讨论打印格式。

所有的函数都返回在一次特定调用中生成的文本字符的计数值。其中两个函数 sprintf 和 vsprintf 将生成的字符保存在一个以空字符结束的字符串 dest 中。因为无法将字符串的最大长度告诉这些打印函数以进行检查，所以用户必须对格式和转换后的数据有足够的了解，以确保字符串能够放进所提供的存储空间。剩余的 4 个函数把数据写入流中。（那些没有 stream 参数的函数把数据写到流 stdout 中。）如果为流的任何一个写操作设置了错误指示符，那么它们将返回一个负值而不是字符的累加计数值。

函数 fprintf、printf 和 sprintf 接受一个可变参数表。当然，这些附加的参数主要用于传递需要转变成文本的数据。为了得到最大的可移植性，必须通过包含 <stdio.h> 来声明这些函数。

vfprintf
vprintf
vsprintf

这 3 个函数很灵活，有时不好区分。C 程序员偶尔需要有细微区别的打印函数。此时就可以选用 vfprintf、vprintf 和 vsprintf 这 3 个函数。除了接收附加参数的方式以外，每个函数的行为与其对应的名字中没有 v 开头的函数的行为一样。使用 <stdarg.h> 中定义的宏来编写一个接受可变参数表的包装函数。这些附加参数被传入打印函数中，以执行实际的转换和文本生成。

例如，如果需要将一些格式化信息写到 stderr 中，且每条信息前都加一个标准前缀，还需要将每一条错误信息都记录到一个磁盘文件中，那么就可以编写函数 eprint 来实现所有这些，其中该函数使用 vfprintf 来执行实际的输出。

```
#include <stdio.h>
#include <stdarg.h>

int eprint(const char *format, ... )
    {  /* log error messages */
    extern FILE *logfile;
    int n;
    va_list ap;

    va_start(ap,format);
    fprintf(stderr,"\aERROR:");
    vfprintf(stderr, format, ap);
    va_start(ap, format);
    n=vfprintf(logfile, format, ap);
    va_end(ap);
    return(n);
    }
```

打印格式

每个打印函数调用的主体部分是为它指定的格式串。在一个微型编程语言中可以（而且应该）把格式串作为程序来考虑。打印函数通过从开始到结尾对格式串进行一次扫描来解释地执行这个程序。当它识别出格式串的每个组成部分时，就会执行各种操作。大多数操作都生成一些字符，该函数将这些字符写入流中或存储到内存中。

大多数操作都需要将参数值转变成字符序列。每个这样的参数都必须按照格式串需要它们的顺序出现在可变参数表中。例如：

```
printf("%s%c%o%i", "th", 'x', 9, 38);
```

经过 4 次转换（th|x|11|38）之后产生字符串 thx1138。程序员必须确保实际的参数表达式的类型与打印函数期望的类型相匹配，因为标准 C 无法检测可变参数表中附加参数的类型。

一定要牢记附加参数遵循的类型转换规则，与在原型声明范围之外调用的函数的参数的相同。例如一个 *float* 类型的参数转换成一个 *double* 类型的参数，一个 *char* 或是 *short* 类型的参数转换成一个 *int* 类型的参数。打印函数可以根据需要对参数进行强制类型转换以便将它们的范围限制到某个特定转换所期望的范围。只有指定一个超出最终类型范围的参数值时，才可能会看到这种机制。例如，转换说明符 %c 期望一个 *int* 类型的参数，这个参数将被转换为 *unsigned char* 类型。所以表达式 printf("%c",0x203) 仅把 3 写入标准输出流。

打印字面量文本

并不是格式串的每一部分都需要附加参数的转换。事实上，只有某些转换说明才针对所有的参数。每一个转换说明都以转义字符百分号 % 开头并且和下面给出的模式之一相匹配。打印函数把格式串中的其他所有部分都

当作字面量文本来对待。字面量文本的每一个字符生成输出的一个字符。

严格地讲，一个格式串是一个多字节字符串。它允许在输出转换中添加日文中的汉字（或者阿拉伯文，或其他任意字符）。如果执行环境对多字节字符使用一个独立于状态的编码，那么每一个字面量文本的序列必须以初始的转移状态开始和结束。（参考第 13 章。）

转换说明

为了构建一个有效的转换说明，% 后面要跟 4 个组成部分。除了最后一个部分，其他的都是可选的。

- 零个或者更多的标志，用来说明标准转换中的变化。
- 一个可选的字段宽度，用来指定转换生成的字符的最小数目。
- 一个可选的精度，控制某些转换生成的字符的数目。
- 一个转换说明符，用来确定参数的类型、参数转换后的值的类型和转换方式。

标志

要按照上面的顺序书写这些组成部分。下面分别详细地讨论每一部分。可以指定以下 5 种不同的标志。

- 一个负号（-），表示将一个转换左对齐，右边填充空格。例如 %-30s。
- 一个零（0），如果没有指定其他的填充字符，就用 0（在任何符号或者前缀之后）填充起始的部分。例如 %04x。
- 一个正号（+），当一个正的有符号的数被转换时，产生一个显式的加号。例如 %+5d。
- 一个空格，当一个正的有符号值被转换时，产生一个空格来代替符号。例如 % 5d。
- 一个 # 号，改变某些转换的行为。o 转换在前面加上一个 0，x 转换在前面加上一个 0x，浮点数转换，生成一个小数点，即使后面没有小数部分。例如 %#x。

字段宽度

字段宽度被表示为一个无符号的十进制整数。写一个星号，然后打印函数就把下一个 *int* 参数作为字段宽度对待，一个负值被当作负标志对待。一个生成小于字段宽度个字符的转换会被填充。在没有负或者零标志的情况下，左边用空格填满。例如 %10c 和 %*i。

精度

精度的表示是以句点（.）开头，后跟一个无符号十进制整数。单独一个句点指精度为零。如果是一个句点后跟一个星号，打印函数就把下一个 *int* 参数当作精度，一个负值当作零对待。精度指定了：

- 转换一个整数时生成的数字的最小位数；
- e、E 或者 f 转换生成的小数数字的位数；
- g 或者 G 转换生成的有效数字的最大位数；
- s 转换生成的字符的最大数目。

例如 %.10e 和 %.*s。

打印转换说明符

可以用一个或者两个字符的序列来表示一个转换说明符，该序列来自一个预先定义好的包含 30 多个有效序列的表。双字符的序列以一个 h、l 或者 L 开头来说明可替代的转换类型。下面列出了所有的有效序列。如果希望代码可以移植，就不要使用其他任何序列。

每一个格式化输出转换的目标都是生成一个文本序列，这个文本序列在转换前完全可以表示编码的值。不幸的是，关于如何“完全地”表示（即使是一个简单的整数值）存在很多不同的观点。这也是存在这么多不同的书写转换说明的方式的原因。对很多转换来说，“完全地”的意思就是“精确地”。但是对浮点转换来说，任何文本表示都可能只是原始值的一个近似值。可以指定保留多少位精度，也可以保证值的符号和数量级可以被正确表示。然而，如果把文本串转换为它的原始编码类型，就不要希望它可以得到完全相同的值。

下面是各种各样的转换说明符。记住，每一个转换都服从填充规则，就像上面为标志和字段宽度所描述的那样。如果没有指定精度 p，那么假设它为规定的默认值。

字符

- c——把 *int* 参数转换为 *unsigned char* 参数来生成一个字符。

十进制数

- d——把 *int* 参数转换为一个至少 p 个十进制数的有符号序列，默认精度是 1。
- hd——把 *int* 参数转换为 *short* 参数，然后和 d 相同。
- ld——*long* 参数的转换和 d 相同。

浮点数

- e——把 *double* 参数转换为一个形如 d.ddde±dd 的有符号序列。这里，d 代表一个十进制位，± 或者是一个正号（+），或者是一个负号（-），点是当前区域设置的小数点。如果 p 是零而且没有指定 # 标志，它就会省略小数点。它生成 p 个小数位和至少两个指数位。默认的精度是 6。
- Le——对 *long double* 参数作和 e 相同的转换。
- E——对 *double* 参数作和 e 相同的转换，只不过把指数前面的 e 换为 E。
- LE——对 *long double* 参数作和 E 相同的转换。
- f——把 *doulbe* 类型的参数转换为一个形式为 d.ddd 的有符号序列。这里，d 代表一个十进制位，小数点是当前区域设置的小数点。它生成至少一个整数位。如果 p 为零而且没有指定 # 标志，它就省略小数点。它生成 p 个小数位。默认的精度是 6。

- Lf——对 *long double* 参数作和 f 相同的转换。
- g——对 *double* 类型的参数作和 e 或者 f 相同的转换。如果 p 没有被指定或者为 0，则把 p 设置为 6。如果 e 形式会产生一个闭区间 [-4,p-1] 内的指数，则选择 f 形式。省略小数后面所有的零。如果没有小数位而且没有指定 # 标志，那么就省略小数点。
- Lg——对 *long double* 参数作和 g 相同的转换。
- G——对 *double* 参数作和 g 相同的转换，只是把任意指数前的 e 替换为 E。
- LG——对 *long double* 类型参数作和 G 相同的转换。

十进制

- i、hi、li——分别和 d、hd、ld 相同。

字符计数

- n——在指向 *int* 类型的指针所指向的数据对象中存储生成字符的累积数目。
- hn——参数为指向 *short* 类型的指针，用法和 n 相同。
- ln——参数为指向 *long* 类型的指针，用法和 n 相同。

无符号整数

- o——把 *int* 参数转换为 *unsigned int* 参数，然后转换为至少 p 个八进制位的无符号序列。默认的精度是 1。
- ho——把 *int* 参数转换为 *unsigned short* 参数，其余操作和 o 相同。
- lo——对 *long* 参数类型作和 o 相同的转换。

指向 *void* 的指针字符串

- p——把指向 *void* 参数的指针转换为一个由实现定义的字符序列（例如一个存储地址的十六进制表示）。
- s——为每一个存储在指向 *char* 参数的指针指向的串中的字符（非空）产生一个字符。如果指定了精度，则最多生成 p 个字符。

无符号整数

- u——把 *int* 参数转换为 *unsigned int* 参数，然后转换为至少有 p 个十进制位的无符号序列，默认的精度是 1。
- hu——把 *int* 参数转换为 *unsigned short* 参数，其余操作和 u 相同。
- lu——把 *long* 参数转换为 *unsigned long* 参数，其余操作和 u 相同。

十六进制

- x——把 *int* 参数转换为 *unsigned int* 参数，然后转换为一个至少有 p 个十六进制位的无符号序列。它用 a 到 f 的字母来表示数字 10 到 15 的数字。默认的精度是 1。
- hx——把 *int* 参数转换为 *unsigned short* 参数，其余操作和 x 相同。
- X——对 *int* 参数作和 x 相同的转换，只不过它用 A 到 F 的字母来表示 10 到 15 的数字。
- hX——把 *int* 参数转换为 *unsigned short* 参数，其余操作和 X 相同。
- lX——把 *long* 参数转换为 *unsigned long* 参数，其余操作和 X 相同。

百分号

- %——不转换任何参数，生成一个百分号字符。

转换说明符能够处理大部分格式化需要。当它们不能满足需求的时候，可以通过两步得到想要的格式。首先，使用 sprintf 在缓冲区生成一个文本并在那里对它进行修改；然后，使用函数，例如 printf，输出这个文本。可以参考图 6-1 的函数 _Fmtval，这是一个实际的例子。

格式化输入

并不是所有的程序都读取输入。那些可以读取输入的程序可以使用标准库函数直接读取数据，然后把数据解释为它们认为合适的形式。为程序的内部使用而转换小整数和文本串是大部分C程序员很容易做的工作。所以，只有当必须转换浮点值或者识别一个复杂的数据域集合的时候，那些标准扫描函数才可能更多地被使用。

即使在这种情况下，是否选择使用标准扫描函数也不是很明确。一个程序的可用性很大程度上决定于它对用户输入的多样性的承受能力。一个程序员可能不同意标准格式化输入函数强迫的转换，也可能不喜欢它们处理错误的方式。总之，他们可能更喜欢使用他们自己的输入方式。

获得格式化输入并不是简单地执行产生格式化输出的反操作。通过输出，程序员可以知道程序下一步要做什么，而且它确实这么做了。然而，通过输入，程序员更多地是被输入那些文本的人所控制。程序必须*扫描*输入文本的一切可识别的样式，然后把它分解为单独的几个域。只有这个时候，程序才知道下一步做什么。

不仅如此，输入文本可能包含不可识别的样式，这时必须确定对这样的“错误”作出怎样的反应。是输出一条错误信息然后提示重新输入？还是提前作一些可能的猜测然后鲁莽前行？还是直接中止程序的执行？各种各样的成品输入扫描器已经尝试过了所有这些策略，但是没有一个能适应所有的情况。

因此，C语言格式化输入函数的历史要比格式化输出函数复杂得多。大部分C实现对printf函数族的基本属性很早以前就达成了共识。相比之下，scanf函数族这些年来则一直在改变并且版本越来越多。X3J11委员会不得不花费很长时间来整理格式化输入的恰当行为。

扫描函数

*扫描*函数被这样称呼的原因是所有的函数都把scan作为它们名字的一部分。这些函数用来扫描输入文本并把文本域转换为编码的数据，它们都在<stdio.h>中声明。要使用扫描函数，必须知道怎样调用它们、怎样指定转换格式和它们会为你实现什么样的转换。C标准库提供了3个不同的扫描函数，声明如下：

```
int fscanf(FILE *stream, const char *format, ... );
int scanf(const char *format, ... );
int sscanf(char *src, const char *format, ... );
```

函数fscanf从流stream中得到字符。函数scanf从流stdin中得到字符。如果获得字符的操作遇到了流的文件结束标志或者错误说明符，这两个函数都会提前停止扫描输入。函数sscanf从以空字符结尾的字符串里获得字符，该字符串是从src开始的。当函数遇到串的结束的空字符时，它就会提前停止扫描输入。

和打印函数一样，所有的扫描函数都接受可变参数表。并且和打印函数一样，在使用任何扫描函数之前，最好通过包含 <stdio.h> 来对它们进行声明。

所有的函数都接受一个 format 参数，这个参数是一个指向可读的且以空字符结尾的串的指针。格式告诉函数所需要的附加参数（如果有的话）和怎样把输入域转换为要存储的值。（一个典型的参数是一个指向接收转换值的数据对象的指针。）它也指定转换域之间想要匹配的字面量文本或空白。如果扫描格式看起来和打印格式非常相似，那么这种相似很有可能是有意的。但是它们之间也存在很多重要的区别。下面会非常详细地讨论扫描格式。

所有的扫描函数都返回一个文本域转换为存储值的数目的计数值。然而，如果任何一个函数因为上面列举的某个原因而提前停止扫描，它就返回 <stdio.h> 中定义的宏 EOF 的值。因为 EOF 一定是一个负值，所以可以很容易地把它和任何有效的计数和零区别开。然而，却不能知道在停止扫描前存储了多少个值。如果需要对停止扫描点进行更精确的定位，那么可以把扫描调用分解为多个调用。

扫描函数也可能在遇到一个不知道怎么处理的字符时中止扫描。在这种情况下，函数返回被转换和存储的值的总数。对任何给定的调用，可以通过对格式中指定的所有转换进行计数来确定最大可能的返回值。实际的返回值会在零和这个最大值之间，包含零和这个最大值。

字符的回退

当 fscanf 或者 scanf 获得一个期望之外的字符时，它把这个字符退回到输入流。（当它必须向前访问来确定域的末端时，它也要把有效域之外的第一个字符回退。）它这样做的方法和调用函数 ungetc 很相似。然而，两者仍然存在一个很重要的区别。不能通过对 ungetc 的连续调用（并且流中没有其他的中间操作）来方便地把两个字符回退到流中。对于相同的流，可以方便地在调用一个任意的扫描函数之后调用 ungetc 函数。

这就意味着，ungetc 带来的单字符回退的限制不能通过调用扫描函数来折中。或者实现保证两个或者更多的字符回送到流中，或者它为扫描函数提供独立的机制。

扫描函数最多回退一个字符。例如，假设想把无效域 123EASY 作为浮点值转换。甚至子域 123E 也是无效的，因为这种转换要求至少有一个指数位。子域 123E 被处理，然后转换就会失败。没有值被存储且扫描函数返回。从流中读取的下一个字符是 A。这种行为对浮点域影响最大，因为浮点数的语法最复杂。其他的转换通常可以消化那些看起来有效的最长子域的所有字符。

扫描格式 前面，我把打印格式描述成一个小型的程序设计语言。当然，扫描格式也是如此。前面我也说过，打印格式和扫描格式非常相似。这句话应该起到一把双刃剑的作用。好的一方面是，打印和扫描函数设计的初衷是在一起工作。在一个程序中写到一个文本文件的内容应该可以作为一个文本文件被另一程序读取。通过调用一个打印函数在文本中表示的任何值都应该可以通过调用一个扫描函数来获取。（至少在一个合理的取值范围内它们的精确度应该很高。）甚至可以让打印和扫描格式彼此非常相似，也有可能写出对称的格式，即可以读取输出值的格式。然而，这要花费一点额外的功夫。

扫描格式与打印格式 这里隐藏着危险。事实是，打印和扫描格式语言是有区别的。有时候，明显的相似性只是表面现象。可以通过调用打印函数来输出文本，但是其扫描方式与使用相同的格式调用扫描函数时的扫描方式不同。当使用没有空白的转换进行文本输出时要特别注意这一点。通过两次连续调用打印函数来打印邻接的空白时也要小心。扫描函数可能会把程序员认为独立的域放在一起处理。

扫描函数的基本操作确实和打印函数的相同。调用一个扫描函数，它就从头到尾对格式串扫描一次。当它识别出格式串的每一个组成部分的时候，它就执行各种操作。大部分操作都顺序使用了流（fscanf 或者 scanf）或者存储在内存中的串（sscanf）中的字符。

很多操作都产生了一些值，扫描函数把这些值存储在各种用指针参数所指定的数据对象中。这样的参数都必须按照格式串需要它们的顺序出现在可变参数表中。例如：

```
sscanf("thx 1138", "%s%2o%d", &a, &b, &c);
```

是把指向串“thx”的指针存储在 *char* 数组 a 中，把值 11 存储在 *int* 数据对象 b 中，把值 38 存储在 *int* 数据对象 c 中。用户要保证每一个实际的参数指针的类型和扫描函数希望的类型匹配，因为标准 C 不能检测可变参数表中的附加参数的类型。

并不是格式串的每一部分都要求域的转换和附加参数的使用。事实上，只有某些**转换说明**才使用参数。每一个转换说明都以转义字符 % 开头，并且和下面给出的其中一种模式匹配。扫描函数把其他所有的东西都作为空白字符或者字面量文本对待。

扫描空白 扫描函数中的空白包含 <ctype.h> 中声明的函数 isspace 所能判断的所有字符。如果调用了 <locale.h> 中声明的函数 setlocale，这些字符会改变。在 "C" 区域设置中，空白就是我们已经知道且喜欢的那些字符。（参考第 2 章。）

12

扫描函数把扫描格式中的一个或者多个空白字符的一个序列作为一个实体对待。这样的序列使扫描函数读取并丢弃输入的任意长度的空白字符的序列。格式中的空白不需要和输入中的相同，输入也可以不包含空白。格式中的空白只是为了保证下一个输入字符（如果有的话）不是一个空白字符。

扫描字面量文本

格式中的任何非空白字符和不是转换说明符的一部分的字符都要求文字匹配。下一个输入字符必须和格式字符匹配。否则，扫描函数返回当前存储的转换值的个数。一个以文字匹配结束的格式可能会产生模糊的结果。我们不能从返回值确定是否存在尾部的不匹配。同样，也不能确定是否发生了文字匹配错误或者它后面有没有转换。由于这些原因，在扫描格式中要限制文字匹配的使用。

文字匹配可以是多字节字符的任意串。如果执行环境对多字节字符使用状态独立编码，那么每一个字面量文本的序列都要以它的初始转移状态开始和结束。（参考第 13 章。）

扫描转换说明

扫描转换说明和打印转换说明的基本方式不同。扫描转换说明符不能编写任何打印转换标志和精度（小数点后面）。相反，它有一个*赋值取消*标志和一个叫做*扫描集*的转换说明。% 后面可以按照以下顺序写 3 个组成部分，除了最后一部分，其他的都是可选的。

赋值取消

- 可以用可选的星号（*）来指定*赋值取消*——不存储转换的值。例如 `%*s`（它会跳过任意的非空白字符序列）。

字段宽度

- 在确定转换域的时候，使用可选的*字段宽度*来指定要匹配的输入字符的最大数目。字段宽度是一个无符号十进制整数。它不限制开头的任意空白的数量。例如 `%5i`。

扫描转换说明符

- 使用*转换说明符*来确定参数的类型、确定转换域的方式和要存储的转换值的方式。可以在括号（`[]`）之间使用*扫描集*转换说明符。所有其他的都由一个预定义的表中的一个或者两个字符的序列组成，这个表包含了 30 多个有效的序列。两个字符的序列以 `h`、`l` 或者 `L` 开头，来说明可替代的参数类型。下面就对扫描集进行说明，也列出了所有的有效序列。如果想让代码可移植，在扫描格式中就别使用任何其他的东西。

每一个格式化输入转换的作用是确定组成要转换的域的输入字符序列。如果可能的话，扫描函数之后就会转换这个域，并且把转换后的值存储在下一个指针参数指向的数据对象中。（如果赋值被取消，就不处理函数的任何参数。）

除非下面有具体说明，否则每一次转换首先跳过输入中的任意的空白。跳过空白的方式跟扫描格式中处理空白的方式一样。然后转换根据随后输入字符的匹配模式来确定转换域。可以指定一个字段宽度来限制域的大小。否则，这个域将一直延伸到与选定模式匹配的最后一个输入字符。

扫描数字域

扫描函数通过调用 F1BS<stdlib.h> 中声明的 C 标准库函数 strtod、strtol、或 strtoul 来转换数字域。一个数字转换域与最长可接受的模式匹配。

下面的描述中简要地概括了每一个有效的转换说明符的匹配模式和转换规则。w 代表指定的字段宽度或是没有指定字段宽度时的默认值。ptr 代表要使用的可变参数列表中的下一个参数。

字符

- c——将 w 个字符（默认值为 1）存储在指针 ptr 所指的 *char* 类型的数组中。它不跳过开头的空白。

十进制数

- d——以 10 为基数，通过调用函数 strtol 来转换整型输入域，然后将结果存放在指针 ptr 所指的 *int* 类型的变量中。
- h——转换方式与 d 一样，但结果存储在 *short* 类型的变量中。
- ld——转换方式与 d 一样，但结果存储在 *long* 类型的变量中。

浮点数

- e——通过调用函数 strtod 来转换浮点型输入域，然后将结果存放在指针 ptr 所指的 *float* 类型的变量中。
- le——转换方式与 e 一样，但结果存储在 *double* 类型的变量中。
- Le——转换方式与 e 一样，但结果存储在 *long double* 类型的变量中。
- E、lE、LE——做法分别与 e、le 和 Le 相同。
- f、lf、Lf——做法分别与 e、le 和 Le 相同。
- q、lq、Lq——做法分别与 e、le 和 Le 相同。
- G、lG、LG——做法分别与 e、le 和 Le 相同。

普通整数

- i——以 0 为基数，通过调用函数 strtol 来转换整数输入域，然后将结果存储在指针 ptr 所指的 int 类型的变量中。（输入时要以 0、0x 或是 0X 开头来指定实际的基数。）
- hi——转换方式和 i 相同，结果存储在 *short* 类型的变量中。
- li——转换方式和 i 相同，结果存储在 *long* 类型的变量中。

字符计数

- n——不转换输入，但是将与输入字符匹配的累计值存储在指针 ptr 所指的 *int* 类型的变量中。
- hn——转换方式与 n 相同，结果存储在 *short* 类型的变量中。
- ln——转换方式与 n 一样，结果存储在 *long* 类型的变量中。

八进制数

- o——以 8 为基数，通过调用函数 strtoul 来转换整型输入，然后将结果存储在指针 ptr 所指的 *unsigned int* 类型的变量中。
- ho——转换方式与 o 一样，但结果存储在 *unsigned short* 类型的变量中。
- lo——转换方式与 o 一样，但结果存储在 *unsigned long* 类型的变量中。

12

指向 *void* 的指针

- p——转换指针输入域，然后将结果存储在指针 ptr 所指的指向 *void* 的指针中。每一个实现都定义了它的指针输入域以和打印函数写入的指针保持一致。

字符串

- s——将最多 w 个非空白字符存储在指针 ptr 所指的 *char* 类型的数组中。它首先跳过开头的空白，最后总是在所有输入之后存储一个空字符。

无符号十进制数

- u——以 10 为基数，通过调用函数 strtoul 来转换整型输入域，然后将结果存储在指针 ptr 所指的 *unsigned int* 类型的变量中。
- hu——转换方式与 u 一样，结果存储在 *unsigned short* 类型的变量中。
- lu——转换方式与 u 一样，结果存储在 *unsigned long* 类型的变量中。

十六进制数

- x——以 16 为基数，通过调用函数 strtoul 来转换整型输入域然后将结果存储在指针 ptr 所指的 *unsigned int* 类型的变量中。
- hx——转换方式与 x 一样，但结果存储在 *unsigned short* 类型的变量中。
- lx——转换方式与 x 一样，但结果存储在 *unsigned long* 类型的变量中。
- x、hX、lX——做法分别与 x、hx 和 lx 相同。

百分号

- %——不转换输入，与一个百分号字符（%）匹配。

扫描集

扫描集的行为与 s 转换说明符的很相像，它将最多 w 个字符（默认是剩余的输入）存储在指针 ptr 所指的 *char* 类型的数组中，而且总是在所有输入之后存储一个空字符。它不跳过开头的空白，也允许指定可以作为域的一部分的字符。可以指定匹配的所有字符，例如 %[0123456789abcdefABCDEF]，它与一个任意的十六进制数的序列相匹配。或者可以指定所有不匹配的字符，例如 %[^0123456789]，它与任意的非数字字符相匹配。

如果想在指定的字符集合中包含右括号（]），则可以直接将右括号写在打开的 [或（[^）的后面，就像 %[][] 可以扫描方括号。不能将空字符包含在指定的字符集合中。某些实现可能允许用减号（-）来指定字符区间。例如，十六进制数字表可以写为 %[0-9abcdefABCDEF]，甚至在某些情况下写为 %[0-9a-fA-F]。然而，这种用法是不通用的。如果希望程序具有最大的可移植性，就要避免使用它。

扫描函数的局限性

扫描转换说明没有打印转换说明那么完备。用户经常希望对输入扫描进行更多的控制，或者有时不能确定一个扫描失败的具体位置，所以就不能从失败中适当地恢复。可以通过与增加打印函数一样的方法来弥补这些不足。首先，将希望扫描的数据读到一个缓冲区中。（有时甚至可以用可以容忍的格式输入，例如 "%s"。）然后使用函数 sscanf 重复扫描缓冲区直到找到一个成功的匹配或是确定了输入错误的属性。然而，当超过某一点的时候扫描函数就很难再正确执行了。而且多年的实践证明它们的用途还是有限的。

下面对 <stdio.h> 中的每一个名字做一些简单的说明。

BUFSIZ BUFSIZ——这个宏产生流缓冲区的首选大小。它的典型范围是从几百到一千多个字节。可以使用 setvbuf 来把该宏作为声明的任意缓冲区的大小使用。

EOF EOF——这个宏用于标志文件的结束。尽管它是一个负值，但是甚至 <ctype.h> 中声明的函数都把它作为参数值来接受它。<stdio.h> 中声明的一些函数也把它作为一个错误返回值来使用。很多实现选择 -1 作为 EOF 的值，但并不是所有的实现都这样做。

FILENAME_MAX FILENAME_MAX——这个宏定义了足够存放任意文件名的字符缓冲区的长度。使用它来声明或是分配这样的缓冲区。在某些系统中，它的长度可以是数百个字节。

FOPEN_MAX FOPEN_MAX——这个宏说明了程序至少可以同时打开的文件数。那 3 个标准 I/O 流都包含在内。例如，可以把这个值用在要创建许多中间临时文件的程序中，这样就可以在新建任何文件之前计划文件的使用。每一个实现必须保证至少可以同时打开 8 个文件。这表明一个可移植的程序最多可以同时打开 5 个附加文件。

_IOFBF _IOFBF——使用这个宏作为 setvbuf 的 mode（第三个）参数来表示完全缓冲。

_IOLBF _IOLBF——使用这个宏作为 setvbuf 的 mode（第三个）参数来表示行缓冲。

_IONBF _IONBF——使用这个宏作为 setvbuf 的 mode（第三个）参数来表示没有缓冲。

L_tmpnam L_tmpnam——这个宏定义了足够存放一个临时文件名的字符缓冲区的长度。使用它来声明或是分配这样的缓冲区。在某些系统中，它的长度可以是数百个字节。

NULL NULL——参考 11.3 节。

SEEK_CUR SEEK_CUR——将这个宏作为 fseek 的 mode（第三个）参数来表明相对于当前的文件定位符的一个定位。对于文本文件，这种模式只对零偏移量有效，它不执行任何操作。

SEEK_END SEEK_END——将这个宏作为 fseek 的 mode（第三个）参数来指示相对于文件结束的一个定位。记住，一个二进制文件可能有附加的空字符，所以这种模式存在不确定的结果。对于文本文件，可以用这种模式来指定没有偏移量。

SEEK_SET SEEK_SET——将这个宏作为 fseek 的 mode（第三个）参数来指示相对于文件开始的一个定位。对于文本文件，偏移量必须是零，或是对于同一个数据流上一次调用函数 ftell 的返回值。

TMP_MAX TMP_MAX——这个宏指明函数 tmpnam 开始循环之前，至少可以生成的不同文件名的个数。例如，可以将这个值用在要创建许多临时中间文件的程序中，这样就可以在新建任何文件之前计划文件的使用。每一实现必须保证至少可以生成 25 个不同的文件名。

stderr stderr——用这个宏来指明标准错误数据流。

stdin stdin——用这个宏来指明标准输入流。

stdout stdout——用这个宏来指明标准输出流。

FILE FILE——可以声明一个指向 FILE 的指针来存储成功调用函数 fopen 或 freopen 的返回值。然后可以将这个值作为对数据流进行操作的各个函数的参数。然而，程序员不能声明 FILE 类型的数据对象。C 标准库提供了所有这样的对象。要把 FILE 类型的数据对象的内容作为一个黑箱子对待，使用 <stdio.h> 中声明的函数来对它的内容进行操作。

fpos_t fpos_t——这是函数 fgetpos 的返回值类型。它可以代表任意文件的任意文件定位符。这说明可以复制这个值，也可以把它作为函数调用中的一个参数来传递，但是不能对它进行算术运算。将这个值传给函数 fsetpos 以便在保存的位置点上重新定位文件。注意，较老的函数 ftell 和 fseek 能够执行相同的操作，但对某些文件（尤其是大文件）也可能失败。无论在什么情况下尽可能使用函数 fgetpos 和 fsetpos。

size_t size_t——参考 11.3 节。

clearerr clearerr——使用这个函数来清除流中的文件结束符和错误指示符。当且仅当使用函数 feof 或 ferror 时才需要使用这个函数。

fclose fclose——如果通过调用函数 fopen 打开了一个文件，那么随后就可能调用函数 fclose 来关闭它。一个对任意数目文件进行操作的程序可能超过允许同时打开的文件数目的最大值。（参见上面讲的 FOPEN_MAX。）程序终止时，C 标准库将关闭仍然处于打开状态的所有文件。这是关闭 3 种标准流的常用方式。

feof feof——大多数读取流的函数会返回一个特定的值，例如 EOF，来指示读操作已到达文件结尾。如果错过了这个测试的机会，则使用函数 feof。它会报告一个流的文件结束符的状态。如果这个指示符显示地将文件定位符从文件结尾移去，那么文件定位请求命令将会清除这个指示符。对于函数 clearerr 的调用也是如此。

ferror ferror——对一个流的读或写操作会因为各种原因而失败。流中的错误指示符将会记录所有这些失败。要检查是否有错误发生，需调用函数 ferror。调用函数 clearerr 或 rewind 将会清除这个指示符。

fflush fflush——可以对一个流调用函数 fflush 来确保流中没有保留缓冲输出。如果向一个输出流写提示信息且从一个输出流读取响应，这个操作可能很重要。要确保与用户知道程序期望的下一个动作是什么。调用函数 fflush（NULL）来清空所有的输出流，这也为程序随后的失去控制做好了准备。（这个程序可能即将执行未调试的代码，或者可能仅仅是引导使用者关闭计算机。）在程序终止时 C 标准库清空所有的输出流。

fgetc fgetc——调用这个函数从输入流中获得下一个字符（参考本节前面“函数 fgetc”部分）。所有读取流的函数的行为就像他们连续调用函数 fgetc 来获得每一个字符。getc 与 fgetc 有相同的说明，但是它可能有一个可以极大地改善性能的屏蔽宏。因此，通常应该使用 getc 而不用 fgetc。

fgetpos fgetpos——使用这个函数来记录文件中以后可能想返回的位置。它返回一个 fpos_t 类型的值，该类型在上面有描述。

fgets fgets——使用这个函数从一个流中读取文本行。当读入并存储了一个换行符或指定的缓冲区已满时，它就停止读取。任何成功读取之后，缓冲区的内容是以空字符结尾的。不要使用函数 gets 来代替这个函数。

fopen fopen——用这个函数打开一个文件。从本节的“打开文件”部分开始就详细讨论了这个函数。使用函数 freopen 来重定向一个标准流。

fprintf fprintf——这是一个格式化输出函数，它将输出内容写到指定的输出流中。参考从本节的“格式化输出”部分开始的描述。

fputc fputc——调用这个函数将一个字符写到输出流中（参考本节的“函数 fputc”部分）。对一个流进行写操作的所有其他函数的行为就像他们可以调用函数 fputc 来传递每一个字符。putc 与 fputc 有相同的说明，但是它更可能有一个可以极大地改善性能的屏蔽宏。因此，通常应该使用 putc 而不是 fputc。

fputs fputs——使用这个函数将字符从以空字符终止的字符串写到一个流中。与函数 puts 不同，函数 fputs 不会将一个换行符附加到它写的任何内容后面。这对汇编文本行或写二进制数据更有用。

fread fread——使用这个函数将二进制数据读入一个数组数据对象中或是从任意数据流中读取最多某个固定数目的字符。如果参数 size（第二个）比 1 大，那么就不能确定这个函数在它报告的字符之外是否也读取了多达 size–1 个额外的字符。通常用 fread(buf,1,size*n,stream) 来调用函数比用 fread(buf,size,n,stream) 更好。

freopen freopen——只有当要重新使用一个已经打开的数据流时才使用函数 freopen。例如，在某些情况下，它可以很方便地将 stdin 或 stdout 重新关联到一个不同的文件。然而，大多情况下都会使用函数 fopen。

fscanf fscanf——这是格式化输入函数，它从指定的输入流中读入数据。参考从本节“格式化输入”部分开始的描述。

fseek fseek——使用这个函数来修改一个数据流的文件定位符。可以通过执行 offset=ftell(stream) 来记录文件中的一个位置。后面可以通过执行 fseek (stream,offeet, SEEK_CUR) 来返回那个位置。fseek 在二进制流中用途更大，在这种情况下，参数 offset（第二个）是文件中的一个 *long* 类型且以字节为单位的偏移量。参数 mode（第三个）必须是上述描述的 SEEK_CUR、SEEK_END 或 SEEK_SET 中的一个。

fsetpos fsetpos——用这个函数为一个流修改文件定位符。它的参数 position（第二个）必须指向一个 fpos_t 类型的数据对象，而该类型是对同据流前面调用函数 fgetpos 时设置的。参考上面对 fpos_t 的讨论。

ftell ftell——用这个函数记录文件中以后想要恢复的位置。它返回一个 *long* 类型的值，该值适用于以后对函数 fseek 的调用。

fwrite fwrite——用这个函数将一个数组数据对象中的二进制数据或者一个定长的字符串写到任意流中。如果参数 size（第二个）的值大于 1，则无法确定在写出错之前该函数是否写了多达 size-1 个额外的字符。写出错通常极少发生，所以这不是一个主要的缺陷。

getc getc——用这个函数代替函数 fgetc，参考函数 fgetc。

getchar getchar——这是函数 getc(stdin) 的一种便利的简写方式。这两个函数的调用生成等价的代码。

gets gets——避免使用这个函数，因为无法限制该函数读入字符的数目。用函数 fgets 代替它。

perror perror——用这个函数向标准错误流写单行出错信息，该信息描述了存储在 errno 中的当前出错代码。（参考第 3 章。）如果想对出错信息的显示方式进行更多的控制，则调用 <string.h> 中声明的函数 strerror 来代替它。

printf printf——这是格式化输出函数，它向标准输出流写数据。该函数是使用最广泛的打印函数，参考上面的函数 fprintf。

putc putc——用这个函数代替函数 fputc。（见 fputc。）

putchar putchar——putchar(ch) 是函数 putc(ch,stdout) 的一种便利的简写方式，这两个函数调用生成等价的代码。

puts puts——用这个函数将字符从一个以空字符终止的字符串写到一个流中。该函数将一个换行符附加到它写的任意数据中。如果不希望附加换行符，则用函数 fputs。

remove remove——该函数实现从一个文件系统中删除一个文件。以后用同一文件名对函数 fopen 调用将无法找到存在的文件。它是删除通过调用函数 tmpnam 生成的文件名新建的任意文件的好方式。

rename rename——这个函数实现文件的重命名。后面对函数 fopen 的调用将找不到一个以旧文件名命名的文件，但能够成功找到一个以新文件名命名的文件。有时可以简单地通过重命名一个临时文件来实现对它的永久保持。然而，rename 不会复制文件内容来使重命名生效。要通过检测函数返回值来看操作是否成功。

rewind rewind——不同于其他文件定位函数，函数 rewind 为一个流清除错误指示符，它也报告没有错误。如果需要，可以使用函数 fseek(stream,0,SEEK_SET) 和 clearerr(stream) 代替它。

scanf scanf——这是格式化的输入函数，它从标准输入流中读取数据。该函数是使用最广泛的输入函数。

setbuf setbuf——用 setvbuf 代替这个函数以获得更多的控制。

setvbuf setvbuf——通常，最好让 C 标准库决定如何缓冲输入 / 输出。如果确定不使用缓冲或一次一行进行缓冲，则用这个函数对流进行合适的初始化。打开某个流之后立即调用函数 setvbuf。几乎对流的任意操作都会抢在用户之前选择一个缓冲策略。如果用户在这次调用中指定了自己的缓冲区，流可能不会真正去使用它。而且当流打开时，绝不要改变缓冲区的内容。参数 mode(第三个) 必须是上述 _IOFBF、_IOLBF 以及 _IONBF 中的一个。也可参考上面描述的宏 BUFSIZ。

sprintf sprintf——这是格式化的输出函数，它将一个以空字符终止的字符串写到指定的缓冲区中。它是允许将已编码的值转换成文本而不用写到一个流的唯一方式。注意不能直接限定函数 sprintf 存储的最大字符数。转换时要小心，因为它可能生成足够的字符而使存储超出了缓冲区的边界。参考函数 fprintf。

sscanf sscanf——这是格式化的输入函数，它从指定的缓冲区中读取一个以空字符终止的字符串。可以使用它以多种不同的格式扫描同一字符序列，直到找到一个成功的扫描。

tmpfile tmpfile——在任何情况下尽可能用函数 tmpfile 代替函数 tmpnam。前者用来打开一个文件并在程序终止时关闭和删除该文件。后者要求手动实现更多的功能。

tmpnam tmpnam——只有当函数 tmpfile 不能满足用户需求时，才用这个函数获得一个或更多的临时文件名。例如，想以一种不同于 "wb+" 的模式打开文件。也可能需要不断地打开和关闭同一个文件。或是想在程序终止之前给文件重命名。参考上述的宏 TMP_MAX。

ungetc ungetc——这个函数只能与读函数结合使用。函数 ungetc 与文件定位函数的相互作用是微弱的。可以使一个不是最后读入的字符回退，甚至可以使文件开始的一个字符回退。但是不能在各个读函数的调用之间方便地回退多个字符。

vfprintf vfprintf——用这个函数构建函数 fprintf 的特殊版本，本节前面有描述。

vprintf vprintf——用这个函数构建函数 printf 的特殊版本，本节前面有描述。

vsprintf vsprintf——用这个函数构建函数 sprintf 的特殊版本，本节前面部分有描述。

12.4 <stdio.h> 的实现

有两种设计决策对于 <stdio.h> 的实现非常关键：

- 数据结构 FILE 的内容；
- 与操作系统相互作用以执行实际输入 / 输出的低级原语。

这一部分先详细讨论这两个话题，然后再讨论低级 I/O 函数的工作方式，最后描述格式化输入输出函数。

头文件 <stdio.h>

图 12-2 显示了文件 stdio.h。现在读者应该能够熟练使用内部头文件 <yvals.h> 来支持依赖实现的参数。

下面是 <yvals.h> 中定义的会影响 <stdio.h> 的一些参数，给这些参数赋予了某些合理的值：

```
#define _NULL    (void *)0   /* value for NULL */
#define _FNAMAX 64  /* value for FILENAME_MAX */
#define _FOPMAX 32  /* value for FOPEN_MAX */
#define _TNAMAX 16  /* value for L_tmpnam */
```

类型 FILE

文件 stdio.h 包含其他一些神秘的东西，现在是揭开它们面纱的时候了。首先集中讨论一下类型 FILE 的定义。它的成员包括以下几个。

- _Mode——流的状态位集合，下面有它的定义。
- _Handle——操作系统为打开文件返回的句柄或文件描述符。
- _Buf——指向流缓冲区首地址的指针，如果没有分配缓冲区，则是一个空指针。
- _Bend——指向超出缓冲区的第一个字符的指针，如果 _Buf 是一个空指针，则它没有定义。
- _Next——指向读操作或写操作的下一个字符的指针，它不可能是一个空指针。
- _Rend——指向超出读取数据范围的第一个字符的指针，它不可能是一个空指针。
- _Rsave——如果字符回退，则保存指针 _Rend。
- _Wend——指向超出可写数据范围的第一个字符，它不可能是一个空指针。
- _Back——保存回退字符的栈。
- _Cbuf——当没有其他缓冲区可用时，能够使用的单字符缓冲区。
- _Nback——记录回退字符个数。
- _Tmpnam——指向文件关闭时要删除的临时文件名的指针，或一个空指针。

getc putc

数据结构 FILE 的设计要受到宏 getc 和 putc（且 getchar 与 putchar）的需要的驱动。这些宏都展开为一个条件表达式，这个表达式可以直接访问流缓冲区或者调用底层函数。条件表达式的判断部分必须简单并且总能够安全执行。这样，如果可读入的字符在指针 str 所指的流的缓冲区中，那么 str->_Next<str->_Rend 总是为真。另外，如果可在缓冲区内获得空间将

字符写到流中，那么 str->_Next<str->_Wend 总是为真。例如，形如 str->_Wend=str->_Buf 的表达式不允许将数据从这些宏写到缓冲区内。

进行读或写流操作时调用的函数构建了更详细的测试。例如，一个读函数能区分多种条件，例如：字符是可用的、当前缓冲区已耗尽、到达文件结束处、还没有分配缓冲区、当前不允许读并且禁止读操作。这些函数很大程度上依靠成员 _Mode 中的各种指示符来进行区分。

头文件 "xstdio.h"

只有 C 标准库内部的函数才关心这些指示符的含义。由于这个以及其他的原因，我创建了内部头文件 "xstdio.h"。这章描述的所有函数都包括 "xstdio.h"，它为流模式指示符定义了宏。该内部头文件包括了 <stdio.h> 并声明了用于实现 <stdio.h> 的属性的所有内部函数，同时定义了许多宏和只与格式化输入输出函数有关的类型。

模式指示符

与 <stdio.h> 不同，头文件 "xstdio.h" 包含了太多琐碎的东西，不能在这里一一介绍。我会在用到它的地方对它进行说明，然后在图 12-57 给出了它的完整文件。例如，这里给出了成员 _Mode 中各种指示符的宏名。每个宏被定义成一个具有不同位集的值，就像在 0x1、0x2、0x4、0x8 中等等。因为实际的值不重要，所以在这里省略了这些值。

- _MOPENR——若文件是打开用来读的，则设置它。
- _MOPENW——若文件是打开用来写的，则设置它。
- _MOPENA——若所有的写操作添加到文件结尾，则设置它。
- _MTRUNC——若已存在的文件在打开（打开后不使用）时被截短了，则设置它。
- _MCREAT——若一个新的文件在打开（打开后不使用）时可以被创建，则设置它。
- _MBIN——若流是二进制的则设置它，若流被解释为文本则不要设置它。
- _MALBUF——若缓冲区在关闭时必须释放，则设置它。
- _MALFIL——若 FILE 类型的数据对象在关闭时必须释放，则设置它。
- _MEOF——文件结束指示符。
- _MERR——错误指示符。
- _MLBF——若行缓冲有效，则设置它。
- _MNBF——若不出现缓冲，则设置它。
- _MREAD——若在上一个文件定位操作之后出现了一个读操作，则设置它。
- _MWRITE——若在上一个文件定位操作之后出现了一个写操作，则设置它。

这些宏都有私有的名字——以一个下划线和一个大写字母开头——尽管不一定要这样写。当我开发库的时候，我经常将它们移入移出 <stdio.h>。某些对用户程序可见的宏使用了这些宏名，但后来的版本又不使用了。最后，为了安全，我将这些宏名保留为这种形式。用户可能偶尔会引入操作成员 _Mode 中的指示符的宏。

图 12-2
stdio.h

```
/* stdio.h standard header */
#ifndef _STDIO
#define _STDIO
#ifndef _YVALS
#include <yvals.h>
#endif
        /* macros */
#define NULL        _NULL
#define _IOFBF      0
#define _IOLBF      1
#define _IONBF      2
#define BUFSIZ      512
#define EOF         -1
#define FILENAME_MAX   _FNAMAX
#define FOPEN_MAX      _FOPMAX
#define L_tmpnam       _TNAMAX
#define TMP_MAX        32
#define SEEK_SET    0
#define SEEK_CUR    1
#define SEEK_END    2
#define stdin       _Files[0]
#define stdout      _Files[1]
#define stderr      _Files[2]
       /* type definitions */
#ifndef _SIZET
#define _SIZET
typedef _Sizet size_t;
#endif
typedef struct {
    unsigned long _Off;                    /* system dependent */
    } fpos_t;
typedef struct {
    unsigned short _Mode;
    short _Handle;
    unsigned char *_Buf, *_Bend, *_Next;
    unsigned char *_Rend, *_Rsave, *_Wend;
    unsigned char _Back[2] , _Cbuf, _Nback;
    char *_Tmpnam;
    } FILE;
        /* declarations */
void clearerr (FILE *);
int fclose(FILE *);
int feof (FILE *);
int ferror(FILE *);
int fflush(FILE *);
int fgetc(FILE *);
int fgetpos(FILE *, fpos_t *) ;
char *fgets (char *, int, FILE *) ;
FILE *fopen(const char *, const char *);
int fprintf (FILE *, const char *, . . .);
int fputc (int, FILE *);
int fputs(const char *, FILE *);
size_t fread (void *, size_t , size_t , FILE *);
FILE *freopen(const char *, const char *, FILE *);
int fscanf(FILE *, const char *, . . . ) ;
```

图 12-2（续）

```
int fseek(FILE *, long, int);
int fsetpos(FILE *, const fpos_t *);
long ftell(FILE *);
size_t fwrite(const void *, size_t, size_t, FILE *);
int getc(FILE *);
int getchar(void);
char *gets(char *);
void perror(const char *);
int printf(const char *,...);
int putc(int, FILE *);
int putchar(int );
int puts(const char *);
int remove(const char *);
int rename(const char *, const char *);
void rewind(FILE *);
int scanf (const char *,...);
void setbuf(FILE *, char *);
int setvbuf(FILE *, char *, int, size_t);
int sprintf(char *, const char *, ...);
int sscanf(const char *, const char *, ...);
FILE *tmpfile(void);
char *tmpnam(char *);
int ungetc(int, FILE *);
int vfprintf(FILE *, const char *, char *);
int vprintf(const char *, char *);
int vsprintf(char *, const char *, char *);
long _Fgpos(FILE *, fpos_t *);
int _Fspos(FILE *, const fpos_t *, long, int);
extern FILE *_Files[FOPEN_MAX];
        /* macro overrides */
#define fgetpos (str, ptr) (int)_Fgpos(str, ptr)
#define fseek(str, off, way) _Fspos(str , _NULL, off, way)
#define fsetpos(str, ptr) _Fspos(str, ptr, 0L, 0)
#define ftell(str) _Fgpos(str, _NULL)
#define getc(str) ((str)->_Next < (str)->_Rend \
   ? *(str)->_Next++ : (getc)(str))
#define getchar() (_Files[0]->_Next < _Files[0]->_Rend \
   ? *_Files[0]->_Next++ : (getchar)())
#define putc(c, str) ((str)->_Next < (str)->_Wend \
   ? (*(str)->_Next++ = c) : (putc)(c, str))
#define putchar(c) (_Files[1]->_Next < _Files[1]->_Wend \
   ? (*_Files[1]->_Next++ = c) : (putchar)(c))
#endif
```
□

这些指示符实际上是两个集合的并集，一个是决定如何打开文件的指示符的集合，另一个是帮助记录数据流状态的指示符的集合。由于这两个集合有部分重叠，所以我选择把它们全部放在一个位编码“空间”中。一个更整洁的实现可能会把这两种用法分开。如果 _Mode 中的位特别少，也会定义两个数值的集合。无论哪种情况，都必须添加代码以在两种表示之间进行转换。

12

函数 fopen

观察库如何使用一个 FILE 类型数据对象的最好方法是在它的整个生命期中跟踪它。图 12-3 显示了文件 fopen.c，它定义了函数 fopen，可以通过文件名调用它来打开一个文件。该函数首先寻找一个空闲的项，该项存储在元素为 FILE 类型指针的静态数组 _Files 中，它包含了 FOPEN_MAX 个元素。如果所有这些指针指向的都是已打开的文件的 FILE 类型的数据对象，那么随后的所有打开请求都会失败。

图 12-3 fopen.c

```
/* fopen function */
#include <stdlib.h>
#include "xstdio.h"

FILE *(fopen)(const char *name, const char *mods)
    {                                           /* open a file */
    FILE *str;
    size_t i;

    for(i = 0; i < FOPEN_MAX; ++i)
       if(_Files[i] == NULL)
          {                         /* setup empty _Files[i] */
          str = malloc (sizeof (FILE));
          if (str == NULL)
              return (NULL) ;
          _Files[i] = str;
          str->_Mode = _MALFIL;
          break;
          }
       else if (_Files[i]->_Mode == 0)
           {                 /* setup preallocated _Files[i] */
           str = _Files[i];
           break;
           }
    if (FOPEN_MAX <= i)
        return(NULL);
    return (_Foprep(name, mods, str));
    }
```
□

数据对象 _Files

图 12-4 显示了定义 _Files 类型数据对象的文件 xfiles.c。该对象为 3 个标准流定义了 FILE 类型数据对象的静态实例。每个实例通过合适的参数被初始化为打开状态。在处理过程中，我用 0 代表标准输入，1 代表标准输出，2 代表标准出错。这是一个被广泛使用的惯例，是从 UNIX 中继承过来的，用户可能需要改变或映射这些值或图。

_Files 中的第四个以及后面的元素被初始化为空指针。如果函数 fopen 发现其中的一个元素，那么它就分配一个 FILE 类型的数据对象并且把它标记为关闭时被释放。函数 fopen 通过观察 _Files 的一个非空元素来发现一个关闭的标准流，这个元素指向成员 _Mode 为零的 FILE 类型数据对象。

图 12-4 xfiles.c

```
/* _Files data object */
#include "xstdio.h"

/* standard error buffer */
static unsigned char ebuf[80];

/* the standard streams */
static FILE sin = {                           /* standard input */
   _MOPENR, 0,
   NULL, NULL, &sin._Cbuf,
   &sin._Cbuf, NULL, &sin._Cbuf,};
static FILE sout = {                          /* standard output */
   _MOPENW, 1,
   NULL, NULL, &sout._Cbuf,
   &sout._Cbuf, NULL, &sout._Cbuf,};
static FILE serr = {                          /* standard error */
   _MOPENW|_MNBF, 2,
   ebuf, ebuf + sizeof(ebuf), ebuf,
   ebuf, NULL, ebuf,};

/* the array of stream pointers */
FILE *_Files[FOPEN_MAX] = {&sin, &sout, &serr};
```

函数 freopen fopen 调用内部函数 _Foprep 来完成打开文件的过程。图 12-5 显示了文件 freopen.c。函数 freopen 也调用这个内部函数。注意在函数 fclose 关闭与当前流关联的文件之前，它是如何记录指示符 _MALFIL 的状态的。函数 freopen 不希望函数 fclose 执行的一个操作是释放 FILE 类型数据对象。

图 12-5 freopen.c

```
/* freopen function */
#include <stdlib.h>
#include "xstdio.h"

FILE *(freopen)(const char *name, const char *mods, FILE *str)
   {                                          /* reopen a file */
   unsigned short mode = str->_Mode & _MALFIL;

   str->_Mode &= ~_MALFIL;
   fclose(str);
   str->_Mode = mode;
   return(_Foprep(name, mods, str));
   }
```

函数 fclose 在这里也会看到 fclose。图 12-6 显示了文件 fclose.c，它以一种非常明显的方式取消文件打开函数所做的工作。其中巧妙的一点是它调用函数 _Fclose 来关闭和流关联的文件。

函数 _Foprep 图 12-7 显示了定义函数 _Foprep 的文件 xfoprep.c。该函数解析 fopen 或 freopen 的参数 mods（第二个），至少是按照它所能理解的，并相应地初始化 FILE 类型的数据对象的成员。然而，最后它必须要求某一外部函数来

完成打开文件的工作。函数 _Foprep 将文件名、编码指示符以及 mods 的值传给调用 _Fopen 的函数，下面非常简短地描述了函数 _Fopen。

图 12-6
fclose.c

```
/* fclose function */
#include <stdlib.h>
#include "xstdio.h"
#include "yfuns.h"

int(fclose)(FILE *str)
   {                                              /*close a stream */
   int stat = fflush(str);

   if(str->_Mode & _MALBUF)
     free(str->_Buf) ;
   str->_Buf = NULL;
   if(0 <= str->_Handle && _Fclose(str))
      stat = EOF;
   if(str->_Tmpnam)
      {                                        /* remove temp file */
      if(remove(str->_Tmpnam))
         stat = EOF;
      free(str->_Tmpnam);
      str->_Tmpnam = NULL;
      }
   str->_Mode = 0;
   str->_Next = &str->_Cbuf;
   str->_Rend = &str->_Cbuf;
   str->_Wend = &str->_Cbuf;
   str->_Nback = 0;
   if(str->_Mode & _MALFIL)
      {                           /* find _Files[i] entry and free */
      size_t i;

      for(i = 0; i < FOPEN_MAX; ++i)
         if(_Files[i] == str)
            {                                     /* found entry */
            _Files[i] = NULL;
            break;
            }
      free(str);
      }
   return(stat);
   }
```

原语 函数 _Fclose 与 _Fopen 是支撑 <stdio.h> 与外部联系的几个低级层次中的第一层，每一个都必须为 C 标准库执行一个标准化函数，也必须便于修改以适应不同操作系统的不同的需要。这个实现包含了 9 个 <stdio.h> 中的函数，它们必须进行修改来适应每一个操作系统。其中有 3 个是标准函数：

- remove——删除一个已命名的文件；
- rename——修改一个文件的名字；
- tmpnam——为一个临时文件构建一个合理的名字。

图 12-7
xfoprep.c

```
/* _Foprep function */
#include "xstdio.h"

/* open a stream */
FILE *_Foprep(const char *name, const char *mods,
    FILE *str)
    {                          /* make str safe for fclose, macros */
    str->_Handle = -1;
    str->_Tmpnam = NULL;
    str->_Buf = NULL;
    str->_Next = &str->_Cbuf;
    str->_Rend = &str->_Cbuf;
    str->_Wend = &str->_Cbuf;
    str->_Nback = 0;
    str->_Mode = (str->_Mode & _MALFIL)
        | (*mods == 'r' ? _MOPENR
        : *mods == 'w' ? _MCREAT|_MOPENW|_MTRUNC
        : *mods == 'a' ? _MCREAT|_MOPENW|_MOPENA
        : 0);
    if((str->_Mode & (_MOPENR|_MOPENW)) == 0)
       {                                        /* bad mods */
       fclose(str);
       return(NULL);
       }                                        /* bad mods */
    while(*++mods== 'b'||*mods == '+')
        if(*mods == 'b')
           if(str->_Mode &_MBIN)
              break;
           else
               str->_Mode |= _MBIN;
       else
           if((str->_Mode & (_MOPENR|_MOPENW))
              ==(_MOPENR|_MOPENW))
              break;
           else
               str->_Mode |= _MOPENR|_MOPENW;
    str->_Handle = _Fopen(name, str->_Mode, mods);
    if(str->_Handle < 0)
       {                                      /* open failed */
       fclose(str);
       return(NULL);
       }
    return(str);
    }
```

这些函数都很小而且很大程度上依赖于底层操作系统的特征，所以不值得把它们作为更低级的原语实现。通常可以在已有的 C 库中找到很多版本，它们使用都很方便。

头文件 "yfuns.h"

有 3 个原语是定义在内部头文件 "yfuns.h" 中的宏，3.4 节中就提到了这个头文件。它定义宏并声明函数，这些宏和函数仅在 C 标准库内部使用，作为与外部联系的接口。只有为这一实现所编写的某些函数需要包括 "yfuns.

h"。(相应地，内部头文件 <yvals.h> 必须包含在几个标准头文件中。) 那 3 个看起来像内部函数的宏声明如下：

```
int _Fclose(FILE *str);
int _Fread(FILE *str, char *buf, int size);
int _Fwrite(FILE *str, const char *buf, int size);
```

它们的语义是：

_Fclose

- _Fclose——关闭与指针 str 关联的文件。如果调用成功，则返回零。

_Fread

- _Fread——从与指针 str 关联的文件中将最多 size 个字符读入从 buf 处开始的缓冲区。返回成功读入的字符数，若在文件结束处则返回零，若发生读错误则返回一个负的错误编码。

_Fwrite

- _Fwrite——将 size 个字符从在 buf 开始的缓冲区写到与指针 str 关联的文件。若调用成功则返回实际写入的字符数，若发生写错误则返回一个负的错误编码。

许多操作系统都支持声明与它们非常相似的函数。存在很多可被宏展开直接调用的函数。

最后 3 个原语是内部函数，其中一个函数在 "xstdio.h" 中声明。其他两个用在屏蔽宏中，因此在 <stdio.h> 中声明。它们的声明如下：

```
short _Fopen(const char *name, unsigned short mode,
   const char *mods);
long _Fgpos(FILE *str, fpos_t *fpos);
int _Fspos(FILE *str, const fpos_t *fpos, long offset, int way);
```

它们的语义如下：

_Fopen

- _Fopen——通过文件名 name 和模式 mode（也可能使用串 mods）打开文件。如果调用成功则返回一个非负的句柄。

_Fgpos

- _Fgpos——如果 fpos 不是一个空指针，就将文件定位符存储在 fpos 中并返回零。否则，将文件定位符编码使之成为一个 *long* 类型的值并返回它的值。若调用失败则返回 EOF 的值。

_Fspos

- _Fspos——若 way 的值为 SEEK_SET，则根据 fpos 或 offset 设置文件定位符。(如果 fpos 不是一个空指针，则使用存储在 fpos 中的值。否则通过给 offset 解码决定文件定位符。) 如果 way 的值为 SEEK_CUR，那么使文件定位符加上 offset 的值。否则 way 的值一定是 SEEK_END，此时将文件定位符设置到文件的最后一个字符之后然后加上 offset。如果成功，则返回零并清除 _MEOF、_MREAD 和 _MWRITE；否则返回值 EOF。

很少能发现一些现有的函数，它们能用来实现这 3 个函数的部分或全部，因为每一个函数都用到了特属于这个实现的数据表示。

附录 A 讨论了这些和其他的接口原语。它讨论了如何将这个库与几个常

用的操作系统配合使用。为了保持完整性，本章给出了一种环境下用到的所有原语。但是请记住，它们只是很多选择中的一种。

UNIX 原语

这里我简单地描述了和 UNIX 操作系统的很多版本通信使用的原语。这通常是作为 C 标准库的宿主环境使用的最简单的系统。即使 C 语言已经移植到了很多其他环境下，但是库的设计的很多方面都是以 UNIX 的需要和功能为原型的。我给出的文件只是最基本的，它们可以在很多方面被扩充。

所有情况下，都假设存在执行 UNIX 系统调用的 C 可调用函数，它们不会和标准 C 的命名空间限制发生冲突。这里使用常规的 UNIX 名字，以一个下划线加一个大写字母开头。因此，unlink 就变成了 _Unlink。如果使用的 UNIX 系统不支持足够的替代函数，则可能需要使用汇编语言编写这些函数。

函数 remove

例如，图 12-8 显示了文件 remove.c，它定义了函数 remove。这个版本只是简单地调用 UNIX 系统调用 _Unlink。一个更严密的版本应该确保具有超级用户权限的程序不会轻易地执行某些操作。

图 12-8 remove.c

```
/* remove function -- UNIX version */
#include "xstdio.h"

        /* UNIX system call */
int _Unlink(const char *);

int(remove)(const char *fname)
   {                                                        /* remove a file */
   return(_Unlink(fname));
}
```

函数 rename

图 12-9 显示了文件 rename.c，它定义了 rename 的一个简单的版本。该函数只执行对文件的连接。这就要求新文件名和旧文件名位于同一个文件系统内（在同一个逻辑磁盘分区上）。一个更好的版本可能会选择在 link 系统服务失败的时候复制一个文件。

图 12-9 rename.c

```
/* rename function -- UNIX version */
#include "xstdio.h"

        /* UNIX system calls */
int _Link(const char *, const char *);
int _Unlink(const char *);

int(rename)(const char *old, const char *new)
   {                                                        /* rename a file */
   return(_Link(old, new) ? -1 : _Unlink(old));
   }
```

函数 tmpnam

图 12-10 显示了文件 tmpnam.c，它定义了 tmpnam 的一个简单的版本。该函数在目录 /tmp 下创建一个临时文件名，这个目录是通常存在临时文件的地方。它对当前的进程 id 进行编码，使每一个控制线程都对应名字列表中的一个唯一的名字。

图 12-10 tmpnam.c

```
/* tmpnam function -- UNIX version */
#include <string.h>
#include "xstdio.h"

        /* UNIX system call */
int _Getpid(void);

char *(tmpnam)(char *s)
    {                               /* create a temporary file name */
    int i;
    char *p;
    unsigned short t;
    static char buf[L_tmpnam];
    static unsigned short seed = 0;

    if(s == NULL)
        s = buf;
    seed = seed == 0 ? _Getpid() : seed + 1;
    strcpy(s, "/tmp/t" );
    i = 5;
    p = s + strlen(s) + i;
    *p = '\0';
    for (t = seed; 0 <= --i; t >>= 3)
        *--p = '0' + (t & 07);
    return (s);
    }
```

函数 _Fopen

图 12-11 显示了文件 xfopen.c，它定义了函数 _Fopen。该函数把模式指示符的编码与打开文件的 UNIX 系统服务使用的编码对应起来。这个程序的正确版本不应该包含这些奇怪的数字，而是应该包含 UNIX 提供的定义相关参数的头文件。

UNIX 不区分二进制和文本文件，其他的操作系统在程序打开文件的时候可能会介意这种差别。同样，UNIX 也不使用附加的模式信息。（这里 _Fopen 把模式参数当作空串对待，因为这个版本并不是特殊的版本。）

函数 _Fgpos

图 12-12 显示了文件 xfgpos.c，它定义了函数 _Fgpos。它要求系统传送文件定位符，然后代表流更正所有缓冲的数据。UNIX 下的文件定位符用 *long* 类型表示。因此，<stdio.h> 中定义的类型 fpos_t 是一个只包含一个 *long* 成员的结构。（我也可以直接把 fpos_t 定义为 *long* 类型，但是我想尽量限制这种类型。）在这种情况下，函数 fgetpos 和 fsetpos 跟旧的文件定位函数相比没有什么优势。然而，对其他系统来说，这种区别还是很重要的。

图 12-11 xfopen.c

```
/* _Fopen function -- UNIX version */
#include "xstdio.h"

        /* UNIX system call */
int _Open(const char *, int, int);

int _Fopen(const char *path, unsigned int smode,
    const char *mods)
    {                                       /* open from a file */
    unsigned int acc;

    acc = (smode & (_MOPENR|_MOPENW)) == (_MOPENR|_MOPENW) ? 2
        : smode & _MOPENW ? 1 : 0;
    if (smode & _MOPENA)
        acc |= 010;                             /* O_APPEND */
    if (smode & _MTRUNC)
        acc |= 02000;                           /* O_TRUNC */
    if (smode & _MCREAT)
        acc |= 01000;                           /* O_CREAT */
    return (_Open(path, acc, 0666));
    }
```

□

图 12-12 xfgpos.c

```
/* _Fgpos function -- UNIX version */
#include <errno.h>
#include "xstdio.h"

        /* UNIX system call */
long _Lseek(int , long, int);

long _Fgpos(FILE *str, fpos_t *ptr)
    {                                       /* get file position */
    long loff = _Lseek(str->_Handle, 0L, 1);

    if(loff ==-1)
       {                                    /* query failed */
       errno = EFPOS;
       return (EOF);
       }
    if(str->_Mode & _MWRITE)
       loff += str->_Next - str->_Buf;
    else if (str->_Mode & _MREAD)
       loff -= str->_Nback
           ? str->_Rsave - str->_Next + str->_Nback
           : str->_Rend - str->_Next;
    if (ptr == NULL)
        return (loff) ;                             /* ftell */
    else
        {                                           /* fgetpos */
        ptr->_Off = loff;
        return (0);
        }
    }
```

□

12

从另一个方面来讲，UNIX 下的 _Fgpos 更加简单。在文本流的内部形式和外部形式之间不发生转换。因此，内部缓冲中对字符的更正也很简单。当然，这是和一个转换文本流的系统相比。比如它用回车加换行来结束每个文本行而不只是一个换行。这就是说，_Fread 必须丢弃某些回车而 _Fwrite 必须插入某些回车；这也意味着 _Fgpos 在更正文件定位符的时候必须纠正某些变更值。这些问题可以处理，但是它们使逻辑混乱，在这里就不进行说明了。

函数 _Fspos

图 12-13 显示了文件 xfspos.c，它定义了函数 _Fspos。与 _Fgpos 一样，它也从简单的 UNIX I/O 模块中受益太多。输出不会造成任何问题，因为函数会在修改文件定位符之前清空所有未写的字符。

图 12-13 *xfspos.c*

```
/* _Fspos function -- UNIX version */
#include <errno.h>
#include "xstdio.h"

      /* UNIX system call */
long _Lseek(int , long, int);

int _Fspos(FILE *str, const fpos_t *ptr, long loff, int way)
   {                                          /* position a file */
   if(fflush(str))
      {                                           /* write error */
      errno = EFPOS;
      return (EOF);
      }
   if (ptr)
       loff += ((fpos_t *)ptr) ->_Off;                /* fsetpos */
   if (way == SEEK_CUR && str->_Mode & _MREAD)
       loff -= str->_Nback
          ? str->_Rsave - str->_Next + str->_Nback
          : str->_Rend - str->_Next;
   if (way == SEEK_CUR && loff != 0
       || way != SEEK_SET || loff != -1)
       loff = _Lseek(str->_Handle, loff, way) ;
   if (loff == -1)
       {                                       /* request failed */
       errno = EFPOS;
       return (EOF);
       }
   else
      {                                               /* success */
      if (str->_Mode & (_MREAD|_MWRITE))
          {                                      /* empty buffer */
          str->_Next = str->_Buf;
          str->_Rend = str->_Buf;
          str->_Wend = str->_Buf;
          str->_Nback = 0;
          }
      str->_Mode &= ~(_MEOF|_MREAD|_MWRITE);
      return (0);
      }
   }
```
□

剩下的 3 个原语都是宏。它们都展开为对直接执行 UNIX 系统服务的函数的调用。"yfuns.h" 的 UNIX 版本包含以下语句：

```
#define _Fclose(str) _Close((str)->_Handle)
#define _Fread(str, buf, cnt) _Read((str)->_Handle, buf, cnt)
#define _Fwrite(str, buf, cnt) _Write((str)->_Handle, buf, cnt)

int _Close(int);
int _Read(int, unsigned char *, int);
int _Write(int, const unsigned char *, int);
```

tmpfile
clearerr
feof
ferror

既然已经明白了 I/O 原语，那么对 <stdio.h> 中声明的大部分低级函数也就容易理解了。现在让我们讨论一下剩下的函数，它们不执行输入输出，只是对流进行设置或者管理。图 12-14 显示了文件 tmpfile.c。函数 tmpfile 是已经遇到的函数中的一个简单应用。图 12-15（clearerr.c）、图 12-16（feof.c）和图 12-17（ferror.c）甚至更简单。这些文件中定义的函数在 <stdio.h> 中没有屏蔽宏，其唯一原因就是它们的使用太少。

图 12-14 tmpfile.c

```
/* tmpfile function */
#include <stdlib.h>
#include <string.h>
#include "xstdio.h"

FILE *(tmpfile)(void)
    {                                       /* open a temporary file */
    FILE *str;
    char fn[L_tmpnam], *s;

    if ((str = fopen((const char *)tmpnam(fn), "wb+")) == NULL)
        ;
    else if ((s = (char *)malloc(sizeof (fn) + 1)) == NULL)
        fclose(str), str = NULL;
    else
        str->_Tmpnam = strcpy(s, fn);
    return (str);
    }
```

图 12-15 clearerr.c

```
/* clearerr function */
#include "xstdio.h"

void (clearerr)(FILE *str)
    {                   /* clear EOF and error indicators for a stream */
    if (str->_Mode & (_MOPENR|_MOPENW))
        str->_Mode &= ~(_MEOF|_MERR);
    }
```

图 12-16 feof.c

```
/* feof function */
#include "xstdio.h"

int (feof)(FILE *str)
    {                   /* test end-of-file indicator for a stream */
    return (str->_Mode &_MEOF);
    }
```

图 12-17
ferror.c

```
/* ferror function */
#include "xstdio.h"

int (ferror)(FILE *str)
    {                               /* test error indicator for a stream */
    return (str->_Mode & _MERR);
    }
```

setbuf
setvbuf

图 12-18 显示了文件 setbuf.c，它简单地由对 setvbuf 的一次调用组成。图 12-19 显示了文件 setvbuf.c。它的大部分工作由清理它的参数组成。注意，setvbuf 是在流没有缓冲的时候响应请求。然而，在发生了任何读或者写操作之后，它不一定能成功执行。

图 12-18
setbuf.c

```
/* setbuf function */
#include "xstdio.h"

void (setbuf)(FILE *str, char *buf)
    {                               /* set up buffer for a stream */
    setvbuf(str, buf, buf ? _IOFBF : _IONBF, BUFSIZ);
    }
```

图 12-19
setvbuf.c

```
/* setvbuf function */
#include <limits.h>
#include <stdlib.h>
#include "xstdio.h"

int (setvbuf)(FILE *str, char *abuf, int smode, size_t size)
    {                               /* set up buffer for a stream */
    int mode;
    unsigned char *buf = (unsigned char *)abuf;

    if (str->_Mode & (_MREAD|_MWRITE))
        return (-1);
    mode = smode == _IOFBF ? 0
        : smode == _IOLBF ? _MLBF
        : smode == _IONBF ? _MNBF : -1;
    if (mode == -1)
        return (-1);
    if (size == 0)
        buf = &str->_Cbuf, size = 1;
    else if (INT_MAX < size)
        size = INT_MAX;
    if (buf)
        ;
    else if ((buf = malloc(size)) == NULL)
        return (-1);
    else
        mode |= _MALBUF;
    if (str->_Mode & _MALBUF)
        free(str->_Buf), str->_Mode &= ~_MALBUF;
    str->_Mode |= mode;
    str->_Buf = buf;
    str->_Bend = buf + size;
    str->_Next = buf;
    str->_Rend = buf;
    str->_Wend = buf;
    return (0);
    }
```

文件定位函数 考虑到原语函数 _Fgpos 和 _Fspos，文件定位函数也是微不足道的。图 12-20 到图 12-24 显示了文件 fgetpos.c、fseek.c、fsetpos.c、ftell.c 和 rewind.c。除了 rewind 函数，我都在 <stdio.h> 中提供了屏蔽宏。

图 12-20 fgetpos.c

```
/* fgetpos function */
#include "xstdio.h"

int (fgetpos)(FILE *str, fpos_t *p)
    {                       /* get file position indicator for stream */
    return (_Fgpos(str, p));
    }
```

图 12-21 fseek.c

```
/* fseek function */
#include "xstdio.h"

int (fseek)(FILE *str, long off, int smode)
    {                                   /* set seek offset for stream */
    return (_Fspos(str, NULL, off, smode));
    }
```

图 12-22 fsetpos.c

```
/* fsetpos function */
#include "xstdio.h"

int (fsetpos) (FILE *str, const fpos_t *p)
    {                       /* set file position indicator for stream */
    return (_Fspos(str, p, 0L, SEEK_SET));
    }
```

图 12-23 ftell.c

```
/* ftell function */
#include "xstdio.h"

long (ftell)(FILE *str)
    {                                   /* get seek offset for stream */
    return (_Fgpos(str, NULL));
    }
```

图 12-24 rewind.c

```
/* rewind function */
#include "xstdio.h"

void (rewind)(FILE *str)
   {                                                   /* rewind stream */
   _Fspos(str, NULL, 0L, SEEK_SET);
   str->_Mode &= ~_MERR;
   }
```

函数 fgetc 现在考虑一下读取字符的函数。图 12-25 显示了文件 fgetc.c，它定义了原型的输入函数 fgetc。该函数首先寻找通过调用 ungetc 函数回退的字符。如果没有，fgetc 就去检查缓冲区中是否有字符。它尝试通过调用 _Frprep 来重新填满一个空缓冲区。如果该函数不能传递任何字符，则 fgetc 返回 EOF。有两个函数是 fgetc 的简单变化形式。图 12-26（getc.c）和图 12-27（getchar.c）都调用 fgetc。

图 12-25
fgetc.c

```
/* fgetc function */
#include "xstdio.h"

int (fgetc)(FILE *str)
    {                                   /* get a character from stream */
    if (0 < str->_Nback)
        {                                  /* deliver pushed back char */
        if (--str->_Nback == 0)
            str->_Rend = str->_Rsave;
        return (str->_Back[str->_Nback]);
        }
   if (str->_Next < str->_Rend)
       ;
   else if (_Frprep(str) <= 0)
        return (EOF);
   return (*str->_Next++);
   }
```

图 12-26
getc.c

```
/* getc function */
#include "xstdio.h"

int (getc)(FILE *str)
    {                                   /* get a character from stream */
    return (fgetc(str));
    }
```

图 12-27
getchar.c

```
/* getchar function */
#include "xstdio.h"

int (getchar)(void)
    {                                    /* get a character from stdin */
    return (fgetc(stdin));
    }
```

函数 ungetc　还有一个函数属于这一组。图 12-28 显示了文件 ungetc.c。前面已经看到了函数 ungetc 对其他几个函数的影响，就是这个函数。考虑到它对其他函数造成的所有影响，函数 ungetc 本身就非常简单了。注意它是怎样修改流的 FILE 数据对象来促使宏 getc 和 getchar 调用它们通常屏蔽的函数的。这使得底层函数有机会让压入到栈中的字符出栈。

fread
fgets
gets　其他几个函数的实现方法跟 fgetc 很像，但由于速度原因，要避免调用 fgetc。一个是 fread，在图 12-29（fread.c）中定义。另外两个在图 12-30（fgets.c）和图 12-31（gets.c）中定义。仔细比较这两个函数，它们的差别很大以致每一个都不值得在另一个的基础上编写。

函数 _Frprep　最后，图 12-32 显示了文件 xfrprep.c，它定义了函数 _Frprep，这个函数执行了读操作的所有严肃的工作。它遇到读错误时返回一个负值，在文件结束处返回零，如果流缓冲现在包含了字符就返回一个正值。这就是分配流缓冲和调用 _Fread 的地方。所有读取流的函数最后都要调用 _Frprep。

图 12-28 ungetc.c

```
/* ungetc function */
#include "xstdio.h"

int (ungetc)(int c, FILE *str)
    {                               /* push character back on stream */
    if (c == EOF
        || sizeof (str->_Back) <= str->_Nback
        || (str->_Mode & (_MOPENR|_MWRITE)) ! = _MOPENR)
        return (EOF);
    str->_Mode = str->_Mode & ~_MEOF | _MREAD;
    if (str->_Nback == 0)
        {                                   /* disable buffering */
        str->_Rsave = str->_Rend;
        str->_Rend = str->_Buf;
        }
    str->_Back[str->_Nback++] = c;
    return ((unsigned char)c);
    }
```

图 12-29 fread.c

```
/* fread function */
#include <string.h>
#include "xstdio.h"

size_t (fread)(void *ptr, size_t size, size_t nelem, FILE *str)
    {                               /* read into array from stream */
    size_t ns = size * nelem;
    unsigned char *s = ptr;

    if (ns == 0)
        return (0);
    if (0 < str->_Nback)
        {                               /* deliver pushed back chars */
        for (; 0 < ns && 0 < str->_Nback; --ns)
            *s++ = str->_Back[--str->_Nback];
        if (str->_Nback == 0)
            str->_Rend = str->_Rsave;
        }
    while (0 < ns)
        {                                   /* ensure chars in buffer */
        if (str->_Next < str->_Rend)
            ;
        else if (_Frprep(str) <= 0)
            break;
         {                          /* deliver as many as possible */
        size_t m = str->_Rend - str->_Next;

        if (ns < m)
            m = ns;
        memcpy(s, str->_Next, m);
        s += m, ns -= m;
        str->_Next += m;
         }
        }
    return ((size * nelem - ns) / size) ;
    }
```

图 12-30
fgets.c

```
/* fgets function */
#include <string.h>
#include "xstdio.h"

char *(fgets)(char *buf, int n, FILE *str)
    {                                       /* get a line from stream */
    unsigned char *s;

    if (n <= 1)
        return (NULL);
    for (s = (unsigned char *)buf; 0 < --n && str->_Nback; )
        {                                   /* deliver pushed back chars */
        *s = str->_Back[--str->_Nback];
        if (str->_Nback == 0)
            str->_Rend = str->_Rsave;
        if (*s++ == '\n')
            {                               /* terminate full line */
            *s = '\0';
            return (buf);
            }
        }
    while (0 < n)
        {                                   /* ensure buffer has chars */
        if (str->_Next < str->_Rend)
            ;
        else if (_Frprep(str) < 0)
            return (NULL);
        else if (str->_Mode & _MEOF)
            break;
         {                                  /* copy as many as possible */
        unsigned char *s1 = memchr(str->_Next,
            '\n', str->_Rend - str->_Next) ;
        size_t m = (s1 ? s1 + 1 : str->_Rend) - str->_Next;

        if (n < m)
            s1 = NULL, m = n;
        memcpy(s, str->_Next, m);
        s += m, n -= m;
        str->_Next += m;
        if (s1)
            {                               /* terminate full line */
            *s = '\0';
            return (buf);
            }
         }
        }
    if (s == (unsigned char *)buf)
        return (NULL);
    else
        {                                   /* terminate partial line */
        *s = '\0';
        return (buf);
        }
    }
```
□

图 12-31 gets.c

```
/* gets function */
#include <string.h>
#include "xstdio.h"

char *(gets)(char *buf)
    {                                         /* get a line from stdio */
    unsigned char *s;

    for (s = (unsigned char *)buf; stdin->_Nback; )
        {                                   /* deliver pushed back chars */
        *s = stdin->_Back[--stdin->_Nback];
        if (stdin->_Nback == 0)
            stdin->_Rend = stdin->_Rsave;
        if (*s++ == '\n')
            {                                    /* terminate full line */
            s[-1] = '\0';
            return (buf);
            }
        }
    for (; ; )
        {                                     /* ensure chars in buffer */
        if (stdin->_Next < stdin->_Rend)
            ;
        else if (_Frprep(stdin) < 0)
            return (NULL);
        else if (stdin->_Mode & _MEOF)
            break;
         {                              /* deliver as many as possible */
        unsigned char *s1 = memchr(stdin->_Next,
            '\n', stdin->_Rend - stdin->_Next);
        size_t m = (s1 ? s1 + 1 : stdin->_Rend)
            - stdin->_Next;

        memcpy(s, stdin->_Next, m);
        s += m; stdin->_Next += m;
        if (s1)
            {                                    /* terminate full line */
            s[-1] = '\0';
            return (buf);
            }
         }
        }
    if (s == (unsigned char *)buf)
        return (NULL);
    else
        {                                     /* terminate partial line */
        *s = '\0';
        return (buf);
        }
    }
```
□

12

图 12-32 xfrprep.c

```
/* _Frprep function */
#include <stdlib.h>
#include "xstdio.h"
#include "yfuns.h"

int _Frprep(FILE *str)
    {                                   /* prepare stream for reading */
    if (str->_Next < str->_Rend)
        return (1);
    else if (str->_Mode & _MEOF)
        return (0);
    else if ((str->_Mode & (_MOPENR|_MWRITE)) != _MOPENR)
        {                               /*can't read after write */
        str->_Mode |= _MERR;
        return (-1);
        }
    if (str->_Buf)
        ;
    else if ((str->_Buf = malloc(BUFSIZ)) == NULL)
        {                               /* use 1-char _Cbuf */
        str->_Buf = &str->_Cbuf;
        str->_Bend = str->_Buf + 1;
        }
    else
        {                               /* set up allocated buffer */
        str->_Mode |= _MALBUF;
        str->_Bend = str->_Buf + BUFSIZ;
        }
    str->_Next = str->_Buf;
    str->_Rend = str->_Buf;
    str->_Wend = str->_Buf;
     {                                  /* try to read into buffer */
    int n = _Fread(str, str->_Buf, str->_Bend - str->_Buf);

    if (n<0)
        {                               /* report error and fail */
        str->_Mode | = _MERR;
        return (-1);
        }
    else if (n == 0)
        {                               /* report end of file */
        str->_Mode = (str->_Mode & ~_MREAD) | _MEOF;
        return (0);
        }
    else
        {                               /* set up data read */
        str->_Mode |= _MREAD;
        str->_Rend += n;
        return (1);
        }
     }
    }
```
□

函数 fputc

下面考虑一下写字符的函数。图 12-33 显示了文件 fputc.c，它定义了原型的输出函数 fputc。该函数首先检查流缓冲是否有空间供写入字符。如果没有可用的空间，函数 fputc 就尝试通过调用 _Fwprep 来设置一个输出缓冲区。如果这个函数不能提供空间，函数 fputc 返回值 EOF。一旦它向缓冲区中添加了一个字符，fputc 在返回之前测试是否用完了缓冲区。有两个函数是 fputc 的简单变化形式。图 12-34（putc.c）和图 12-35（putchar.c）都调用 fputc。

图 12-33 fputc.c

```
/* fputc function */
#include "xstdio.h"

int (fputc)(int ci, FILE *str)
    {                                   /* put a character to stream */
    unsigned char c = ci;

    if (str->_Next < str->_Wend)
        ;
    else if (_Fwprep(str) < 0)
        return (EOF);
    *str->_Next++ = c;
    if (str->_Mode & (_MLBF|_MNBF))
        {                               /* disable macros and drain */
        str->_Wend = str->_Buf;
        if ((str->_Mode & _MNBF || c == '\n') && fflush(str))
            return (EOF);
        }
    return (c);
    }
```

图 12-34 putc.c

```
/* putc function */
#include "xstdio.h"

int (putc)(int c, FILE *str)
    {                                   /* put character to stream */
    return (fputc(c, str));
    }
```

图 12-35 putchar.c

```
/* putchar function */
#include "xstdio.h"

int (putchar)(int c)
    {                                   /* put character to stdout */
    return (fputc(c, stdout));
    }
```

函数 _Fwprep

图 12-36 显示了文件 xfwprep.c，它定义了函数 _Fwprep，这个函数完成写的所有准备工作。发生写错误时，函数返回一个负值；如果流缓冲现在包含可以写入字符的空间，就返回零。这就是流缓冲分配的地方，所有向流写入内容的函数最后都要调用 _Fwprep。

图 12-36
xfwprep.c

```
/*_Fwprep function */
#include <stdlib.h>
#include "xstdio.h"
#include "yfuns.h"

int _Fwprep(FILE *str)
    {                                   /* prepare stream for writing */
    if (str->_Next < str->_Wend)
        return (0);
    else if (str->_Mode & _MWRITE)
        return (fflush(str));
    else if ((str->_Mode & (_MOPENW|_MREAD)) ! = _MOPENW)
        {                               /* can't write after read */
        str->_Mode |= _MERR;
        return (-1);
        }
    if (str->_Buf)
        ;
    else if ((str->_Buf = malloc(BUFSIZ)) == NULL)
        {                               /* use 1-char _Cbuf */
        str->_Buf = &str->_Cbuf;
        str->_Bend = str->_Buf + 1;
        }
    else
        {                               /* use allocated buffer */
        str->_Mode |= _MALBUF;
        str->_Bend = str->_Buf + BUFSIZ;
        }
    str->_Next = str->_Buf;
    str->_Rend = str->_Buf;
    str->_Wend = str->_Bend;
    str->_Mode |= _MWRITE;
    return (0);
    }
```
□

函数 fflush　图 12-37 显示了文件 fflush.c，这就是调用函数 _Fwrite 把流缓冲的内容写出的地方。如果参数是一个空指针，函数就对 _Files 数组的每一个非空元素调用它本身。这里用递归代替循环来使控制流更加清晰。对这样的调用来说，性能不是主要的考虑因素。

函数 perror　还有一个函数也属于这个组。图 12-38 显示了文件 perror.c，它生成一条错误信息并把它写入标准错误流中。函数 _Strerror 完成函数 strerror（这两个函数都在 <string.h> 中声明）的工作，但是使用的缓冲区是调用者提供的。不允许函数 perror 修改 strerror 中的静态存储空间中的内容。因此，每一个函数都应该用它自己的静态缓冲区调用 _Strerror。

fwrite
fputs
puts

其他几个函数的实现方法跟 fputc 很像，但由于速度原因，要避免调用 fputc。fputc 的一个变化形式是 fwrite，在图 12-39（fwrite.c）中定义。另外两个在图 12-40（fputs.c）和图 12-41（puts.c）中定义。后者是前者的一个简单变化形式。

图 12-37
fflush.c

```
/* fflush function */
#include "xstdio.h"
#include "yfuns.h"

int (fflush)(FILE *str)
    {                                        /* flush an output stream
*/
    int n;
    unsigned char *s;

    if (str == NULL)
        {                                    /* recurse on all streams */
        int nf, stat;

        for (stat = 0, nf = 0; nf < FOPEN_MAX; ++nf)
            if (_Files[nf] && fflush(_Files[nf]) < 0)
               stat = EOF;
        return (stat);
        }
    if (! (str->_Mode & _MWRITE))
        return (0);
    for (s = str->_Buf; s < str->_Next; s += n)
        {                                    /* try to write buffer */
        n = _Fwrite(str, s, str->_Next - s);
        if (n <= 0)
            {                                /* report error and fail */
            str->_Next = str->_Buf;
            str->_Wend = str->_Buf;
            str->_Mode |= _MERR;
            return (EOF);
            }
        }
    str->_Next = str->_Buf;
    str->_Wend = str->_Bend;
    return (0);
```

图 12-38
perror.c

```
/* perror function */
#include <errno.h>
#include <string.h>
#include "xstdio.h"

void (perror)(const char *s)
    {                                    /* put error string to stderr */
    static char buf[] = {"error #xxx"};

    if (s)
        {                                /* put user-supplied prefix */
        fputs(s, stderr);
        fputs(": ", stderr);
        }
    fputs(_Strerror(errno, buf), stderr);
    fputc('\n', stderr);
    }
```
□

12

图 12-39
fwrite.c

```
/* fwrite function */
#include <string.h>
#include "xstdio.h"

size_t (fwrite)(const void *ptr, size_t size,
    size_t nelem, FILE *str)
    {                                    /* write to stream from array */
    char *s = (char *)ptr;
    size_t ns = size * nelem;

    if (ns == 0)
        return (0);
    while (0 < ns)
        {                                    /* ensure room in buffer */
        if (str->_Next < str->_Wend)
            ;
        else if (_Fwprep(str) < 0)
            break;
        {                                /* copy in as many as possible */
        char *s1 = str->_Mode & _MLBF
            ? memchr (s, '\n', ns) : NULL;
        size_t m = s1 ? s1 - s + 1 : ns;
        size_t n = str->_Wend - str->_Next;

        if (n < m)
            s1 = NULL, m = n;
        memcpy(str->_Next, s, m);
        s += m, ns -= m;
        str->_Next += m;
        if (s1 && fflush(str))
            {                            /* disable macros on failure */
            str->_Wend = str->_Buf;
            break;
            }
         }
        }
    if (str->_Mode & _MNBF)
        {                                    /* disable and drain */
        str->_Wend = str->_Buf;
        fflush(str);
        }
    return ((size * nelem - ns) / size);
    }
```

□

图 12-40 fputs.c

```
/* fputs function */
#include <string.h>
#include "xstdio.h"

int (fputs)(const char *s, FILE *str)
    {                                       /* put a string to stream */
    while (*s)
        {                                   /* ensure room in buffer */
        if (str->_Next < str->_Wend)
            ;
        else if (_Fwprep(str) < 0)
            return (EOF);
         {                              /* copy in as many as possible */
         const char *s1 = str->_Mode &_MLBF
             ? strchr(s, '\n') : NULL;
         size_t m = s1 ? s1 - s + 1 : strlen(s);
         size_t n;

         n = str->_Wend - str->_Next;
         if (n < m)
             s1 = NULL, m = n;
         memcpy(str->_Next, s, m);
         s += m;
         str->_Next += m;
         if (s1 && fflush(str))
             {                                      /* fail on error */
             str->_Wend = str->_Buf;
             return (EOF);
             }
          }
         }
    if (str->_Mode & _MNBF)
        {                               /* disable macros and drain */
        str->_Wend = str->_Buf;
        if (fflush(str))
            return(EOF);
        }
    return (0);
    }
```

图 12-41 puts.c

```
/* puts function */
#include "xstdio.h"

int (puts)(const char *s)
    {                              /* put string + newline to stdout */
    return (fputs(s, stdout) < 0
        || fputc('\n', stdout) < 0 ? EOF : 0);
    }
```

这就是低级输入输出函数的完整集合。就像你所看到的，没有一个特别难。然而，整个集合却累积了很多代码，而且这仅仅是开始。实现 <stdio.h> 比较困难的部分是执行格式化输入输出。

格式化输出

有 6 个函数执行格式化输出（打印函数）。所有的函数都调用一个公共函数 _Printf，这个函数声明如下：

```
int _Printf(void *(*pfn)(void *, const char *, size_t),
    void *arg, const char *fmt, va_list ap);
```

用到的参数如下：

- pfn——指向一个函数的指针，该函数用来传送字符。
- arg——通用的数据对象指针，作为传送函数的一个参数被传递。
- fmt——指向格式串的指针。
- ap——指向上下文信息的指针，该信息描述了一个可变参数表。

如果成功，传送函数返回 arg 的一个新值；否则，它返回一个空指针来标志一个写错误。

fprintf
printf

图 12-42 显示了文件 fprintf.c，它定义了函数 fpirntf 和它使用的传送函数 prout。在这种情况下，通用指针把 FILE 指针从 fprintf 函数通过 _Printf 传送给 prout。调用 fprintf 时，prout 就使用这个指针对指定的流进行写操作。图 12-43 显示了文件 printf.c，它是 fprintf 的一个简单的变化形式。

图 12-42
fprintf.c

```
/* fprintf function */
#include "xstdio.h"

static void *prout(void *str, const char *buf, size_t n)
    {                                       /* write to file */
    return (fwrite(buf, 1, n, str) == n ? str : NULL);
    }

int (fprintf)(FILE *str, const char *fmt, ...)
    {                              /* print formatted to stream */
    int ans;
    va_list ap;

    va_start(ap, fmt);
    ans = _Printf(&prout, str, fmt, ap);
    va_end(ap);
    return (ans);
    }
```
□

图 12-43 printf.c

```
/* printf function */
#include "xstdio.h"

static void *prout(void *str, const char *buf, size_t n)
    {                                           /* write to file */
    return (fwrite(buf, 1, n, str) == n ? str : NULL);
    }

int (printf)(const char *fmt, ...)
    {                                   /* print formatted to stdout */
    int ans;
    va_list ap;

    va_start(ap, fmt);
    ans = _Printf (&prout, stdout, fmt, ap);
    va_end(ap);
    return(ans);
    }
```

其他打印函数 图 12-44 显示了文件 sprintf.c。这里，当调用 sprintf 时，通用指针指明了在指定的缓冲中存储字符的下一个位置。注意，如果 _Printf 执行成功，函数 sprintf 也写入一个终止的空字符。图 12-45 到图 12-47 显示了文件 vfprintf.c、vprintf.c 和 vsprintf.c。它们都是那 3 个更常用的打印函数的变化形式。

图 12-44 sprintf.c

```
/* sprintf function */
#include <string.h>
#include "xstdio.h"

static void *prout(void *s, const char *buf, size_t n)
    {                                       /* write to string */
    return ((char *) memcpy(s, buf, n) + n);
    }

int (sprintf)(char *s, const char *fmt, ...)
    {                                   /* print formatted to string */
    int ans;
    va_list ap;

    va_start(ap, fmt);
    ans = _Printf(&prout, s, fmt, ap);
    if (0 <= ans)
        s[ans] = '\0';
    va_end(ap);
    return (ans);
    }
```

图 12-45 vfprintf.c

```
/* vfprintf function */
#include "xstdio.h"

static void *prout(void *str, const char *buf, size_t n)
    {                                           /* write to file */
    return (fwrite(buf, 1, n, str) ==n ? str : NULL);
    }

int (vfprintf)(FILE *str, const char *fmt, char *ap)
    {                   /* print formatted to stream from arg list */
    return (_Printf(&prout, str, fmt, ap));
    }
```

图 12-46
vprintf.c

```
/* vprintf function */
#include "xstdio.h"

static void *prout(void *str, const char *buf, size_t n)
    {                                           /* write to file */
    return (fwrite(buf, 1, n, str) == n ? str : NULL);
    }

int (vprintf)(const char *fmt, char *ap)
    {                       /* print formatted to stdout from arg list */
    return (_Printf(&prout, stdout, fmt, ap));
    }
```

图 12-47
vsprintf.c

```
/* vsprintf function */
#include <string.h>
#include "xstdio.h"

static void *prout(void *s, const char *buf, size_t n)
    {                                           /* write to string */
    return ((char *)memcpy(s, buf, n) + n);
    }

int (vsprintf)(char *s, const char *fmt, char *ap)
    {                       /* print formatted to string from arg list */
    int ans = _Printf(&prout, s, fmt, ap);

    if (0 <= ans)
        s[ans] = '\0';
    return (ans);
    }
```

函数 _Printf

图 12-48 显示了文件 xprintf.c，它定义了函数 _Printf，该函数完成所有的工作。<stdlib.h> 中声明的内部函数 _MBtowc，使用用户在每次调用时提供的 _Mbstate 类型的状态记忆，把格式分解为一个多字节串进行分析。（参考第 13 章。）通过调用底层函数而不是 mbtowc，_Printf 就可以避免改变 mbtowc 的内部状态。C 标准禁止任何这样的改变。

测试转义字符百分号（%）是一件很麻烦的事。唯一安全的方式是把格式串转换为一个宽字节字符序列，然后寻找一个和百分号对应的字符。必须把 wc 数据对象和百分号的宽字节字符编码进行对比。不幸的是，那些值存在着某些不确定性。C 标准要求基本 C 字符集中的每一个字符都要有一个宽字节字符编码与它的单字符编码等价。百分号的单字符编码为 '%'，那么等价的宽字节字符编码就是 L'%'。但是，仍然存在一个问题，就是 C 标准是否应该要求这样的等价形式。这样可能会导致用户依靠规则的某个脆弱点轻率地编写代码。

还存在另外一个不确定性。一个实现可以支持宽字节字符的多种编码，至少原则上可以。当宽字节字符常量和当前的字符集不匹配的时候，可以想到一个程序会改变一个区域设置。这可能不是很明智，但 C 标准并没有明确地禁止这一点。因此，一个谨慎的程序应该避免使用 '%' 或者 L'%' 作为百分号的宽字节字符编码。

图 12-48
xprintf.c

```
/*_Printf function */
#include <ctype.h>
#include <stdlib.h>
#include <string.h>
#include "xstdio.h"

#define MAX_PAD (sizeof (spaces) - 1)
#define PAD(s, n) if (0 < (n)) {int i, j = (n); \
   for (; 0 < j; j -= i) \
       {i = MAX_PAD < j ? MAX_PAD : j; PUT(s, i);} }
#define PUT(s, n) \
   if (0 < (n)) {if ((arg = (*pfn)(arg, s, n)) != NULL) \
       x.nchar += (n); else return (EOF); }

static char spaces [] = "                                ";
static char zeroes[] = "00000000000000000000000000000000";

int _Printf(void *(*pfn)(void *, const char *, size_t),
    void *arg, const char *fmt, va_list ap)
    {                                           /* print formatted */
    _Pft x;

    for (x.nchar = 0; ; )
        {                                   /* scan format string */
        const char *s = fmt;

        {                                /* copy any literal text */
        int n;
        wchar_t wc;
        _Mbsave state = {0};

        while (0 < (n = _Mbtowc(&wc, s, MB_CUR_MAX, &state)))
            {                              /* scan for '%' or '\0' */
            s += n;
            if (wc == '%')
                {                  /* got a conversion specifier */
                --s;
                break;
                }
            }
        PUT(fmt, s - fmt);
        if (n <= 0)
            return (x.nchar);
        fmt = ++s;
         }
         {                       /* parse a conversion specifier */
        const char *t;
        static const char fchar[] = {" +-#0"};
        static const unsigned int fbit[] = {
           _FSP, _FPL, _FMI, _FNO, _FZE, 0};
```

12

图 12-48（续）

```
        for (x.flags = 0; (t = strchr(fchar, *s)) != NULL; ++s)
            x.flags |= fbit[t - fchar];
        if (*s == '*')
            {                               /* get width argument */
            x.width = va_arg(ap, int);
            if (x.width < 0)
                {                           /* same as '-' flag */
                x.width = -x.width;
                x.flags |=_FMI;
                }
            ++s;
            }
        else                            /* accumulate width digits */
            for (x.width = 0; isdigit(*s); ++s)
                if (x.width < _WMAX)
                    x.width = x.width * 10 + *s - '0';
        if (*s != '.')
            x.prec = -1;
        else if (*++s == '*')
            {                           /* get precision argument */
            x.prec = va_arg(ap, int);
            ++s;
            }
        else                        /* accumulate precision digits */
            for (x.prec = 0; isdigit(*s); ++s)
                if (x.prec < _WMAX)
                    x.prec = x.prec * 10 + *s - '0';
        x.qual = strchr("hlL", *s) ? *s++ : '\0';
        }
        {                                   /* do the conversion */
        char ac[32];

        _Putfld(&x, &ap, *s, ac);
        x.width -= x.n0 + x.nz0 + x.n1 + x.nz1 + x.n2 + x.nz2;
        if (!(x.flags & _FMI))
            PAD(spaces, x.width);
        PUT(ac, x.n0);
        PAD(zeroes, x.nz0);
        PUT(x.s, x.n1);
        PAD(zeroes, x.nz1);
        PUT(x.s + x.n1, x.n2);
        PAD(zeroes, x.nz2);
        if (x.flags & _FMI)
            PAD(spaces, x.width);
        }
        fmt = s + 1;
    }
}
```
□

实现者对于和 wc 作比较的值有 3 种选择：

- 为了保持和旧的 C 翻译器保持最大的兼容，使用 '%'。依赖于等价的且不随区域设置改变的编码。
- 为了最大的清晰性，使用 L'%'。依赖于不随区域设置改变的编码。
- 在声明了 wchar_t wcs[2] 之后，在 _Printf 的每个入口执行 mbstowcs (wcs,"%",1)。这样就把百分号当前的宽字节字符编码存储在了 wcs[0] 中。(mbstowcs 在 <stdlib.h> 中声明。)

鉴于 C 翻译器、C 标准和多字节字符所支持的当前状态，我选择了第一种最明智的方案。然而，这个领域是飞速发展的。在不久的将来，另外一种选择可能会更加谨慎。

_Printf 的剩余代码或者它的附属函数就不用担心多字节字符的问题了。转换说明符由基本 C 字符集中的字符组成。每一个说明符都有一个单字符编码。(原则上，一个格式串可能在转换说明符中包含多余的转移编码，但我选择了不支持这种行为。)

PUT
PAD

因此 _Printf 只担心转换说明符之间的字面量文本中的多字节字符。一旦它发现了很大一部分字面量文本，它就传送这样的字符直到遇到百分号，但不包括百分号。宏 PUT 定义在这个 C 源文件顶部，用来传送字符。一定要注意对它的使用。不能把这个操作打包成一个函数。如果传送函数报告一个错误，它就需要从 _Printf 中返回。另一方面，重复地使用这样繁琐的功能也没有什么好处。由于很多相同的原因，我也创建了宏 PAD 用来传送填充的零或者空格。

一旦 _Printf 在格式中遇到了百分号，它就分析后面的转换说明符。它将标志转换为函数 _Printf 和它的子函数中使用的指示符集。头文件 "xstdio.h" 包含了下面的宏定义：

```
#define _FSP    0x01
#define _FPL    0x02
#define _FMI    0x04
#define _FNO    0x08
#define _FZE    0x10
```

这些定义分别和空格、+、-、# 和 0 这几个标志一一对应。

宏 _WMAX

头文件 "xstdio.h" 把宏 _WMAX 定义为 999。_Printf 使用这个值来限制字段宽度和精度值的大小。它必须足够大以描述支持的最大的转换（至少生成 509 个字符）；也要足够小以阻止 *short* 整数的溢出（不能超过 32767）。我使用 999 来简化对累加器循环的测试。

类型 _Pft

_Printf 把关于转换说明符的信息放到一个 _Pft 类型的结构 x 中。附属函数填充附加的信息。当它们完成它们的工作时，_Printf 只要测试 x 的内容就可以知道要传送什么字符。头文件 "xstdio.h" 包含了以下类型定义：

```
typedef struct {
    union {
        long li;
        long double ld;
```

```
	} v;
char *s;
int n0, nz0, n1, nz1, n2, nz2, prec, width;
size_t nchar;
unsigned int flags;
char qual;
} _Pft;
```

它的成员有：

- v——在接收参数（_Putfld）的函数和把参数转换为文本（_Litob 或者 _Ldtob）的函数之间传递一个整数值（v.li）或者一个浮点值（v.ld）。
- s——传递 v 的转换使用的文本缓冲的地址。
- n0——对文本缓冲 ac 开始处 _Printf 首先要传送的字符的数目进行计数。
- nz0——对下一次要传送的零的数目进行计数。
- n1——对 ac 中下一次要传送的后续字符的数目进行计数。
- nz1——对下一次要传送的零的数目进行计数。
- n2——对 ac 中下一次要传送的后续字符的数目进行计数。
- nz2——对下一次要传送的零的数目进行计数。
- prec——保存转换说明符中的精度（如果没有的话，值为 -1）。
- width——保存转换说明符中的字段宽度（如果没有的话，值为 0）。
- nchar——对目前为止已经传送的字符进行计数。
- flags——保存转换说明符中编码的标志。
- qual——保存转换说明符中的大小限定符（h、l、L）。

所有的计数器都是有必要的，这样可以把对文本缓冲 ac 的大小的需要减到最小。缓冲应该足够大以表示数字转换中所有有意义的精度，这样才有意义。然而，不需要在缓冲中写入很长的零序列。最好用一个像 PAD 这样的宏来对它们进行计数和生成它们的值。

两个例子可以说明这个问题。第一个是表达式 printf("%015.5f ", -1e4)，它生成文本 -00010000.00000。注意 3 个 0、4 个 0 和 5 个 0 的序列混杂在其他文本之间。这对它们在缓冲中的聚集没什么坏处。但是，如果把表达式改为 printf("%0500.200f",- 1e37) 会发生什么呢？它是一个任意实现必须支持的可移植的表达式。它也生成了数百个零，其中最少的序列有 37 个零，它就需要更大的缓冲了。

为了得到更大的灵活性，我添加了一些复杂的东西，而不是对字段宽度或者精度进行额外的限制。那些转换值的函数的逻辑很难阅读，但换来的是代码可以处理很多苛刻的需求。

函数 _Putfld

图 12-49 显示了文件 xputfld.c，它定义了函数 _Putfld，函数 _Printf 调用它来处理一个转换说明。_Putfld 函数由一个很大的 *switch* 语句组成，它分组处理转换说明符。_Putfld 从可变参数表中搜集需要的参数，然后直接处理数字转换的符号和所有只涉及文本的转换，而把实际的数字转换交给两个附属函数中的一个来处理。

图 12-49 xputfld.c

```
/* _Putfld function */
#include <string.h>
#include "xstdio.h"

        /* macros */
#if _DLONG
#define LDSIGN(x) \
   (((unsigned short *)&(x))[_D0 ? 4 : 0] & 0x8000)
#else
#define LDSIGN(x) (((unsigned short *)&(x))[_D0] & 0x8000)
#endif

void _Putfld(_pft *px, va_list *pap, char code, char *ac)
    {                                   /* convert a field for _Printf */
    px->n0 = px->nz0 = px->n1 = px->nz1 = px->n2 = px->nz2 = 0;
    switch (code)
        {                           /* switch on conversion specifier */
    case 'c':                           /* convert a single character */
        ac[px->n0++]=va_arg(*pap,int);
        break;
    case 'd': case 'i':         /* convert a signed decimal integer */
        px->v.li = px->qual == 'l' ?
          va_arg(*pap, long) : va_arg(*pap, int);
      if (px->qual == 'h')
          px->v.li = (short)px->v.li;
      if (px->v.li < 0)                    /* negate safely in _Litob */
          ac[px->n0++] = '-';
      else if (px->flags & _FPL)
          ac[px->n0++] = '+';
      else if (px->flags & _FSP)
          ac[px->n0++] = ' ';
      px->s = &ac[px->n0];
      _Litob(px, code);
      break;
    case 'o': case 'u':
    case 'x': case 'X':                             /* convert unsigned */
        px->v.li = px->quai == 'i' ?
            va_arg(*pap, long) : va_arg(*pap, int);
        if (px->qual == 'h')
            px->v.li = (unsigned short)px->v.li;
        else if (px->qual == '\0')
            px->v.li = (unsigned int)px->v.li;
        if (px->flags & _FNO && px->v.li != 0)
            {                          /* indicate base with prefix */
            ac[px->n0++] = '0';
            if (code == 'x' || code == 'X')
                ac[px->n0++] = code;
            }
        px->s = &ac[px->n0];
        _Litob(px, code);
        break;
    case 'e': case 'E': case 'f':                /* convert floating */
    case 'g': case 'G':
        px->v.ld = px->qual == 'L' ?
            va_arg(*pap, long double) : va_arg(*pap, double);
```

12

图 12-49（续）

```
        if (LDSIGN(px->v.ld))
            ac[px->n0++] = '-';
        else if (px->flags & _FPL)
            ac[px->n0++] = '+';
        else if (px->flags & _FSP)
            ac[px->n0++] = ' ';
        px->s = &ac[px->n0];
        _Ldtob(px, code);
        break;
    case 'n' :                              /* return output count */
        if (px->qual == 'h')
            *va_arg(*pap, short *) = px->nchar;
        else if (px->qual != 'l')
            *va_arg(*pap, int *) = px->nchar;
        else
            *va_arg(*pap, long *) = px->nchar;
        break;
    case 'p' :              /* convert a pointer, hex long version */
        px->v.li = (long)va_arg(*pap, void *);
        px->s = &ac[px->n0];
        _Litob(px, 'x');
        break;
    case 's':                                   /* convert a string */
        px->s = va_arg(*pap, char *);
        px->n1 = strlen(px->s);
        if (0 <= px->prec && px->prec < px->n1)
            px->n1 = px->prec;
        break;
    case '%':                                          /* put a '%' */
        ac[px->n0++] = '%';
        break;
    default:                  /* undefined specifier, print it out */
        ac[px->n0++] = code;
        }
    }
```
□

_Putfld 通过调用 _Litob 执行所有的整数转换。图 12-50 显示了文件 xlitob.c，它定义了函数 _Litob。它转换的值 px->v.li 类型为 *long*。这有一点冒险。如果把一个大于 LONG_MAX 的 *unsigned long* 类型的值存储在 *long* 类型的变量中，一个计算机体系结构就可以报告算术溢出。因此表达式 printf ("%x", 0x80000000L) 可能会正确地打印，但也可能不会。C 标准规定所有整数转换的参数都应该是有符号类型。因此，这种冒险是由于打印函数遗传的弱点引起的，而不是因为任何实现的某些决策。

从好的一方面说，_Putfld 和 _Litob 都很严密。它们避免了求 *long* 类型的相反数，因为那种操作在 2 的补码机器下会造成溢出。相反，_Putfld 让 _Litob 把值转换为 *unsigned long* 类型，然后再求它的相反数，这样就不会造成溢出了。只要一个任意的 *unsigned long* 类型可以安全地转换为 long 类型然后再转换回来，这个实现就可以很好地工作。很多机器下都是这种情况。

宏 LDSIGN

_Putfld 在测试浮点值时也同样严密。对于像 NaN 或者 Inf 这样的特殊编码值需要特殊的处理方式，以免在 _Putfld 内部发生异常。因此，宏 LDSIGN 使用半数值方法测试 *long double* 的符号位。它效仿了 7.4 节的宏 DSIGN。

图 12-50 xlitob.c

```
/* _Litob function */
#include <stdlib.h>
#include <string.h>
#include "xmath.h"
#include "xstdio.h"

static char ldigs[] = "0123456789abcdef ";
static char udigs[] = "0123456789ABCDEF";

void _Litob(_Pft *px, char code)
    {                               /* convert unsigned long to text */
    char ac[24];                          /* safe for 64-bit integers */
    char *digs == code == 'x' ? udigs : ldigs;
    int base = code == 'o' ? 8 :
        code != 'x' && code != 'X' ? 10 : 16;
    int i = sizeof (ac);
    unsigned long ulval = px->v.li;

    if ((code == 'd' || code == 'i' ) && px->v.li < 0)
        ulval = -ulval;                        /* safe against overflow
*/
    if (ulval || px->prec)
        ac[--i] = digs[ulval % base];
    px->v.li = ulval / base;
    while (0 < px->v.li && 0 < i)
        {                                          /* convert digits */
        ldiv_t qr = ldiv(px->v.li, base);

        px->v.li = qr.quot;
        ac[--i] = digs[qr.rem];
        }
    px->n1 = sizeof (ac) - i;
    memcpy(px->s , &ac[i], px->n1);
    if (px->n1 < px->prec)
        px->nz0 = px->prec - px->n1;
    if (px->prec < 0 && (px->flags & (_FMI|_FZE)) == _FZE
        && 0 < (i = px->width - px->n0 - px->nz0 - px->n1))
        px->nz0 += i;
```

指向 *void* 的指针

一个更有问题的实现决策是关于 p 这个转换说明符，在 C 标准中，它打印 *void* 指针的方式是留给实现定义的。在这个实现中，我选择把这个指针强制转换为指向 *long* 的指针，然后把它作为一个十六进制整数打印出来。然而，指针和整数不同，不能保证这种做法在一个给定的体系结构下正确或者安全。在某些机器下，这里的代码可能必须经过修改才能有效地工作。

_Litob 本身相当地简单。为了安全性，它使用 unsigned long 算术转换一个数字。然后，它使用 *long* 算术转换剩下的所有数字，这样在很多计算机体系结构下执行速度都会很快。这个函数在内部缓冲中从右到左处理数字，然后把它们复制到从 _Printf 继承来的缓冲中。注意函数计算数字前面的零的数目的方法。它保证零的数目至少和要求的精度一样多，但是如果需要左移，可以更多。

函数 _Ldtob

_Putfld 通过调用函数 _Ldtob 来执行所有的浮点转换。图 12-51 显示了文件 xldtob.c，它定义了函数 _Ldtob。它转换的值 px->v.ld 类型为 *long double*，该类型足够表示任何浮点值。

图 12-51
xldtob.c

```
/* _Ldtob function */
#include <float.h>
#include <stdlib.h>
#include <string.h>
#include "xmath.h"
#include "xstdio.h"

        /* macros */
#define NDIG    8

        /* static data */
static const long double pows[] = {
    1e1L, 1e2L, 1e4L, 1e8L, 1e16L, 1e32L,
#if 0x100 < _LBIAS            /* assume IEEE 754 8- or 10-byte */
    1e64L, 1e128L,  1e256L,
#if _DLONG                             /* assume IEEE 754 10-byte */
    1e512L, 1e1024L, 1e2048L, 1e4096L,
#endif
#endif
    };

void _Ldtob(_Pft *px, char code)
    {                               /* convert long double to text */
    char ac[32];
    char *p = ac;
    long double ldval = px->v.ld;
    short errx, nsig, xexp;

    if (px->prec < 0)
        px->prec = 6;
    else if (px->prec == 0 && (code == 'g' || code == 'G'))
        px->prec = 1;
    if (0 < (errx =_Ldunscale(&xexp, &px->v.ld)))
        {                               /* x == NaN, x == INF */
        memcpy(px->s, errx == NAN ? "NaN" : "Inf", px->n1 = 3);
        return;
        }
    else if (0 == errx)                                 / * x == 0 */
        nsig = 0, xexp = 0;
    else
        {                                   /* 0 < |x|, convert it */
         {                          /* scale ldval to ~~10^(NDIG/2) */
        int i, n;

        if (ldval < 0.0)
            ldval = -ldval;
        if ((xexp = xexp * 30103L / 100000L - NDIG/2) < 0)
            {                                       /* scale up */
            n = (-xexp + (NDIG/2-1)) & ~(NDIG/2-1) , xexp = -n;
           for (i = 0; 0 < n; n >>= 1, ++i)
               if (n & 1)
                   ldval *= pows[i];
           }
        else if (0 < xexp)
            {                                       /* scale down */
```

图 12-51 （续）

```
        long double factor = 1.0;

        xexp &= ~(NDIG/2-1);
        for (n = xexp, i = 0; 0 < n; n >>= 1, ++i)
            if (n & 1)
                factor *= pows[i];
        ldval /= factor;
        }
      }
      {                        /* convert significant digits */
    int gen = px->prec
        + (code == 'f' ? xexp + 2+NDIG : 2+NDIG/2);

    if (LDBL_DIG+NDIG/2 < gen)
        gen = LDBL_DIG+NDIG/2;
    for (*p++ = '0'; 0 < gen && 0.0 < ldval; p += NDIG)
        {                            /* convert NDIG at a time */
        int j;
        long lo = (long)ldval;

        if (0 < (gen -= NDIG))
            ldval = (ldval - (long double)lo) *1e8L;
        for (p += NDIG, j = NDIG; 0 < lo && 0 <= --j; )
            {                        /* convert NDIG digits */
            ldiv_t qr = ldiv(lo, 10);

            *--p = qr.rem + '0' , lo = qr.quot;
            }
        while (0 <= --j)
            *--p = '0';
        }
     gen = p - &ac[1];
     for (p = &ac[1], xexp += NDIG-1; *p == '0'; ++p)
         --gen, --xexp;                     /* correct xexp */
     nsig = px->prec + (code == 'f' ? xexp + 1
         : code == 'e' || code== 'E' ? 1 : 0);
     if (gen < nsig)
         nsig = gen;
     if (0 < nsig)
         {                   /* round and strip trailing zeros */
         const char drop
             = nsig < gen && '5' <= p[nsig] ? '9' : '0';
         int n;

         for (n = nsig; p[--n] == drop; )
             --nsig;
         if (drop == '9')
             ++p[n];
         if (n < 0)
             --p, ++nsig, ++xexp;
         }
     }
   }
  _Genld(px, code, p, nsig, xexp);
  }
```
□

函数 _Ldtob 位于 <stdio.h> 和 <math.h> 之间。它包含了头文件 "xstdio.h" 和 "xmath.h" 来获得所有需要的参数。它也共享了 <math.h> 的这个实现中的很多假设。例如，数据对象 pows 包含了所有形式为 10^{2^N} 的可表示的浮点值。我选择了区分以下 3 个区间。

- 最小的区间，最大到 10^{32}。
- IEEE 754 的 8 字节表示，最大到 10^{256}。
- IEEE 754 的 10 字节表示，最大到 10^{4096}。

有时可能需要修改这个表来适应其他实现。

_Ldtob 使用 "xmath.h" 中声明的函数 _Ldunscale 来测试和拆分浮点值。对一个存储在 px->v.ld 中的有限值 x 来说，_Ldunscale 用小数 f 来代替 x，这里 $|f|$ 在半开区间 [0.5,1.0) 内；把指数 e 存储在 xexp 中，这里 $x = f * 2^e$。在这种情况下，_Ldtob 对 f 没有任何作用。它只使用 e 来把 x（现在在 ldval 中）扩大到一个合适的区间。

如果 _Ldunscale 报告 x 为非数，_Ldtob 就生成 NaN。如果 x 是无穷大，函数则生成 Inf。C 标准没有定义遇到非数或者无穷大时会出现什么情况，所以产生这些序列是一个合理的扩充。

_Ldtob 通过把 *long double* 类型的 ldval 的值赋给 *long* 类型的 lo，一次去掉了 8（NDIG）个数字。*long* 类型至少可以表示大到 10^9 的值。通常，把一个 *long* 类型值转换为 8 位十进制数要比转换任何浮点类型快得多。函数也尽量只转换那些转换说明符要求的数字。

要达到转换的这些节省机制需要合理的组织。注意下面这个特殊的赋值语句：

```
xexp = xexp * 30103L / 100000L - NDIG/2;
```

它对 ldval(x) 要求的预引比例因子进行了充分的评估。用户希望乘以 pows 的元素的最小数目，那么最后 ldval 一定要比 10^8 小。用户还希望第一组的 8 个数字至少有 4 个非零数字，那么必须使用实际的比例因子（在 xexp 中）来生成一个合适的指数。这个表达式通过把 e 乘以 $\log_{10}(2)$ 来开始那个过程。它也允许小数点的左面有 4 位数字。函数然后相应地修改 ldval。

下一个特殊的近似值是下面的初始化式：

```
int gen = px-prec
    + (code == 'f' ? xexp + 2+NDIG : 2+NDIG/2);
```

它对要转换的数字的位数进行充分的评估，它允许至少一个额外的数字对结果进行舍入处理。对比之下，后面的转换就相当简单了。转换最后丢弃尾部的零，并相应地调整 gen 和 xexp。

下一步是计算转换说明要求的有效数字 nsig 的位数。（直到获得了 xexp

的准确值之后才可以这样做。）剩下的工作就是把 nsig 缩减到给出的有效数字的数目。如果 nsig 比 gen 小，函数也要对结果进行舍入处理。_Ldtob 最后通过调用函数 _Genld 来结束。这样就省去了对转换后的值进行修改来满足各种转换说明符的具体需要。

函数 _Genld

图 12-52 显示了文件 xgenld.c，它定义了函数 _Genld。该函数在 _Printf 提供的缓冲区中产生各种浮点转换的最后表示。它按照从左到右的顺序，从 _Ldtob 的缓冲中复制需要的字符来完成这项操作。这里的逻辑冗长而严密，但没有什么技巧。有一点特别的是，xexp 会因为 f 转换说明符而改变它的含义。它变成对开头的数字的计数，而不是要显示的指数。类似地，px->prec 为 g 转换说明符改变了它的含义。它变成了小数部分数字的计数，而不是总的精度。

那就是打印函数的最后一部分代码。正如你所看到的，转换浮点值花费了相当多的努力，它也包含了大量的代码。小型计算机下的标准 C 的实现可能不需要打印浮点值。这种情况下，可以通过提供 _Putfld 的一个修改的版本来大幅度地缩减程序的大小。删除浮点转换的代码，这样就不需要再连接 _Ldtob 和它的附属函数，也不用连接很多其他提供浮点支持的函数。

然而，为同一个函数编写多个版本早晚会造成混乱。

格式化输入

有 3 个函数执行格式化输入（扫描函数），它们都调用一个公共函数 _Scanf，这个函数声明如下：

```
int _Scanf(void *(*pfn)(void *, int), void *arg,
    const char *fmt, va_list ap);
```

它的参数包含如下几个：

- pfn——指向函数的指针，调用该函数来获得字符。
- arg——一个通用的数据对象指针，作为那个获取函数的其中一个参数传递。
- fmt——指向格式串的指针。
- ap——指向上下文信息的指针，该信息描述了一个可变参数表。

若获取函数的第二个参数值为 _WANT（在 "xstdio.h" 中定义，是一个与所有字符编码和 EOF 都不同的值），则它就获得下一个要扫描的字符。否则，这个函数就把第二个参数作为要回退的字符对待。函数失败时返回 EOF。

12

图 12-52
xgenld.c

```
/* _Genld function */
#include <locale.h>
#include <string.h>
#include "xstdio.h"

void _Genld(_Pft *px, char code, char *p, short nsig,
    short xexp)
    {                               /* generate long double text */
    const char point = localeconv()->decimal_point[0];

    if (nsig <= 0)
        nsig = 1, p = "0";
    if (code == 'f' || (code == 'g' || code == 'G')
        && -4 <= xexp && xexp < px->prec)
        {                                             /* 'f' format */
        ++xexp;                     /* change to leading digit count */
        if (code != 'f')
            {                                      /* fixup for 'g' */
            if (!(px->flags & _FNO) && nsig < px->prec)
                px->prec = nsig;
            if ((px->prec -= xexp) < 0)
                px->prec = 0;
            }
        if (xexp <= 0)
            {                     /* digits only to right of point */
            px->s[px->n1++] = '0';
            if (0 < px->prec || px->flags & _FNO)
                px->s[px->n1++] = point;
            if (px->prec < -xexp)
                xexp = -px->prec;
            px->nz1 = -xexp;
            px->prec += xexp;
            if (px->prec < nsig)
                nsig = px->prec;
            memcpy(&px->s[px->n1] , p, px->n2 = nsig);
            px->nz2 = px->prec - nsig;
            }
        else if (nsig < xexp)
            {                               /* zeros before point */
            memcpy(&px->s[px->n1], p, nsig);
            px->n1 += nsig;
            px->nz1 = xexp - nsig;
            if (0 < px->prec || px->flags & _FNO)
                px->s[px->n1] = point, ++px->n2;
            px->nz2 = px->prec;
            }
        else
            {                          /* enough digits before point */
            memcpy(&px->s[px->n1], p, xexp);
            px->n1 += xexp;
            nsig -= xexp;
            if (0 < px->prec || px->flags & _FNO)
                px->s[px->n1++] = point;
            if (px->prec < nsig)
                nsig = px->prec;
```

图 12-52
（续）

```
        memcpy(&px->s[px->n1], p + xexp, nsig);
        px->n1 += nsig;
        px->nz1 = px->prec - nsig;
        }
      }
  else
      {                                          /* 'e' format */
      if (code == 'g' || code == 'G')
          {                                    /* fixup for 'g' */
          if (nsig < px->prec)
              px->prec = nsig;
          if (--px->prec < 0)
              px->prec = 0;
          code = code == 'g' ? 'e' : 'E';
          }
      px->s[px->n1++] = *p++;
      if (0 < px->prec || px->flags & _FNO)
          px->s[px->n1++] = point;
      if (0 < px->prec)
          {                         /* put fraction digits */
          if (px->prec < --nsig)
              nsig = px->prec;
          memcpy(&px->s[px->n1], p, nsig);
          px->n1 += nsig;
          px->nz1 = px->prec - nsig;
          }
      p = &px->s[px->n1];                        /* put exponent */
      *p++ = code;
      if (0 <= xexp)
          *p++ = '+';
      else
          {                                 /* negative exponent */
          *p++ = '-';
          xexp = -xexp;
          }
      if (100 <= xexp)
          {                          /* put oversize exponent */
             if (1000 <= xexp)
                 *p++ = xexp / 1000 + '0', xexp %= 1000;
             *p++ = xexp / 100 + '0', xexp %= 100;
       }
       *p++ = xexp / 10 + '0', xexp %= 10;
       *p++ = xexp + '0';
       px->n2 = p - &px->s[px->n1];
       }
  if ((px->flags & (_FMI|_FZE)) == _FZE)
      {                             /* pad with leading zeros */
      int n = px->n0 + px->n1 + px->nz1 + px->n2 + px->nz2;

      if (n <px->width)
          px->nz0 = px->width - n;
      }
  }
```
□

12

图 12-53
fscanf.c

```
/* fscanf function */
#include "xstdio.h"

static int scin(void *str, int ch)
    {                                   /* get or put a character */
    if (ch == _WANT)
        return (fgetc((FILE *)str));
    else if (0 <= ch)
        return (ungetc(ch, (FILE *)str));
    else
        return (ch);
    }

int (fscanf)(FILE *str, const char *fmt, ...)
    {                               /* read formatted from stream */
    int ans;
    va_list ap;

    va_start(ap, fmt);
    ans = _Scanf(&scin, str, fmt, ap);
    va_end(ap);
    return (ans);
    }
```

图 12-54
scanf.c

```
/* scanf function */
#include "xstdio.h"

static int scin(void *str, int ch)
    {                                   /* get or put a character */
    if (ch == _WANT)
        return (fgetc((FILE *)str));
    else if (0 <= ch)
        return (ungetc(ch, (FILE *)str));
    else
        return (ch);
    }

int (scanf)(const char *fmt, ...)
    {                                /* read formatted from stdin */
    int ans;
    va_list ap;

    va_start(ap, fmt);
    ans = _Scanf (&scin, stdin, fmt, ap);
    va_end(ap);
    return (ans);
    }
```

图 12-55 sscanf.c

```
/* sscanf function */
#include "xstdio.h"

static int scin(void *str, int ch)
	{	/* get or put a character */
	char *s = *(char **)str;

	if (ch == _WANT)
		if (*s == '\0')
			return (EOF);
		else
			{	/* deliver a character */
			*(char **)str = s + 1;
			return (*s);
			}
	else if (0 <= ch)
		*(char **)str = s - 1;
	return (ch);
	}

int (sscanf)(const char *buf, const char *fmt, ...)
	{	/* read formatted from string */
	int ans;
	va_list ap;

	va_start(ap, fmt);
	ans = _Scanf(&scin, (void **)&buf, fmt, ap);
	va_end(ap);
	return (ans);
	}
```

函数 _Scanf

图 12-56 显示了文件 xscanf.c，它定义了函数 _Scanf，这个函数完成所有的工作。

类型 _Sft

_Scanf 把各个信息位都放到一个 _Sft 类型的结构 x 中。附属函数填充附加的信息。当它们完成一个给定的转换说明符的所有工作的时候，_Scanf 通过检查 x 的内容，就会知道已经扫描了多少个字符和最后一个转换说明符是否存储了转换后的值。头文件 "xstdio.h" 包含以下的类型定义：

```
typedef struct {
	int (*pfn)(void *, int);
	void *arg;
	va_list ap;
	int nchar, nget, width;
	char noconv, qual, stored;
	}_Sft;
```

它的成员如下：

- pfn——指向获取函数的指针。
- arg——保存获取函数的一般参数。
- ap——保存可变参数表的上下文信息。
- nchar——对目前为止扫描的字符的总数进行计数。

12

图 12-56 xscanf.c

```
/* _Scanf function */
#include <ctype.h>
#include <limits.h>
#include <stdlib.h>
#include <string.h>
#include "xstdio.h"

int _Scanf(int (*pfn)(void *, int), void *arg,
    const char *fmt, va_list ap)
    {                                           /* read formatted */
    const char *s;
    int nconv = 0;
    _Sft x;

    x.pfn = pfn;
    x.arg = arg;
    x.ap = ap;
    x.nchar = 0;
    for (s = fmt; ; ++s)
        {                                   /* parse format string */
        int ch;

         {                    /* match any literal or white-space */
        int n;
        wchar_t wc;
        _Mbsave state = {0};

        while (0 < (n = _Mbtowc(&wc, s, MB_CUR_MAX, &state)))
            {                        /* check type of multibyte char */
            s += n;
            if (wc == '%')
                break;
            else if (wc <= UCHAR_MAX && isspace(wc))
                {                           /* match any white-space */
                while (isspace(*s))
                    ++s;
                while (isspace(ch = GET(&x)))
                    ;
                UNGET(&x, ch);
                }
            else                                /* match literal text */
                for (s -= n; 0 <= --n; )
                    if ((ch = GET(&x)) ! = *s++)
                        {                               /* bad match */
                        UNGET(&x, ch);
                        return (nconv);
                        }
            }
        if (*s == '\0')
            return (nconv);
         }
         {                        /* process a conversion specifier */
```

图 12-56（续）

```
        x.noconv = *s == '*' ? *s++ : '\0';
        for (x.width = 0; isdigit(*s); ++s)
            if (x.width < _WMAX)
                x.width = x.width * 10 + *s - '0';
        x.qual = strchr("hlL". *s) ? *s++ : '\0';
        if (!strchr("cn[", *s))
            {                       /* match leading white-space */
            while (isspace(ch = GET(&x)))
                ;
            UNGET(&x, ch);
            }
        if ((s = _Getfld(&x, s)) == NULL)
            return (0 < nconv ? nconv : EOF);
        if (x.stored)
            ++nconv;
      }
     }
}
```

- nget——对目前为止宏 GETN（下面会讲）扫描的字符的个数进行计数。
- width——保存转换说明的字段宽度（如果没有的话为 0）。
- noconv——保存一个非零值（'*'）来取消存储转换值。
- qual——保存转换说明的大小限定符（h、l、L)。
- stored——通过 _Scanf 的一个存储转换值的附属函数设置为非零。

<stdlib.h> 中声明的内部函数 _Mbtowc，使用用户每次调用时提供的 _Mbstate 类型的状态记忆，把格式作为一个多字节串进行分析。它遇到的问题和 _Printf 相同，12.4 节有相应的描述。然而，_Scanf 不但要区分百分号字符而且要区分空白。假设任何可以以 *unsigned char* 类型存储的宽字节字符编码都可以通过 isspace 得到正确的判断。在当前的 C 标准中这肯定是正确的。但在 '\t' 和 L'\t' 不一定等价的环境下，修改起来就比较麻烦了。

和 _Printf 一样，_Scanf 也要担心转换说明符之间的字面量文本中的多字节字符。一旦它发现了很大一部分字面量文本，它就传送这样的字符直到遇到百分号，但不包括百分号。它匹配空白的方式很有趣。并且只有当扫描的文本和格式中的字面量文本的转移序列相同时，它才匹配多字节字符。这两种特性都会限制扫描函数的功能，但它们又都是遗传的，那就是 C 标准说明扫描函数的方式。

GET
UNGET

宏 GET 用来获得一个字符，宏 UNGET 用来把第一个不想要的字符回退，要注意它们的使用方法。这两个宏都在 "xstdio.h" 中定义，因为 _Scanf 的附属函数也要通过同样的方式获得字符。两个宏的定义如下：

```
#define GET(px)(++(px)->nchar, (*(px)->pfn)((px)->arg,_ WANT))
#define UNGET(px, ch) \
   (--(px)->nchar, (*(px)->pfn)((px)->arg, ch))
```

也可以把这些操作作为函数实现，我把它们定义为宏主要是为了提高性能。

GETN UNGETN

头文件 "xstdio.h" 定义了两个和这些联系紧密的附加的宏。可以把一个字符计数存储在 x.nget 中来定义用户希望扫描的字段的最大宽度。使用宏 GETN 来代替 GET，用 UNGETN 代替 UNGET。一旦字段用完了，GETN 就生成特殊的编码 _WANT，这样就简化了几个地方的逻辑。这两个宏的定义为：

```
#define GETN(px) ( 0 < = -- (px) -> nget ? GET(px) : _WANT)
#define UNGETN(px, ch) {if (ch) != _WANT) UNGET(px, ch); }
```

头文件 "xstdio.h"

这是对头文件 "xstdio.h" 最后的主要贡献。图 12-57 显示了文件 xstdio.h。到了这个时候，它应该已经非常明确了。在这里把它给出来只是为了保持完整性。

图 12-57 xstdio.h

```
/* xstdio.h internal header */
#include <stdarg.h>
#include <stdio.h>
        /* bits for _Mode in FILE */
#define _MOPENR 0x1
#define _MOPENW 0x2
#define _MOPENA 0x4
#define _MTRUNC 0x8
#define _MCREAT 0x10
#define _MBIN   0x20
#define _MALBUF 0x40
#define _MALFIL 0x80
#define _MEOF   0x100
#define _MERR   0x200
#define _MLBF   0x400
#define _MNBF   0x800
#define _MREAD  0x1000
#define _MWRITE 0x2000
        /* codes for _Printf and _Scanf */
#define _FSP    0x01
#define _FPL    0x02
#define _FMI    0x04
#define _FNO    0x08
#define _FZE    0x10
#define _WMAX   999
#define _WANT   (EOF-1)
        /* macros for _Scanf */
#define FMAX   512                    /* widest supported field */
#define GET(px)(++(px)->nchar, (*(px)->pfn)((px)->arg, _WANT))
#define GETN(px)   (0 <= --(px)->nget ? GET(px) : _WANT)
#define UNGET(px, ch) \
   (--(px)->nchar, (*(px)->pfn)((px)->arg, ch))
#define UNGETN(px, ch) {if ((ch) ! = _WANT) UNGET(px, ch); }
        /* type definitions */
typedef struct {
   union {
       long li;
       long double ld;
       } v;
    char *s;
    int n0, nz0, n1, nz1, n2, nz2, prec, width;
    size_t nchar;
    unsigned int flags;
    char qual;
    } _Pft;
typedef struct {
    int (*pfn)(void *, int);
    void *arg;
    va_list ap;
    int nchar, nget, width;
    char noconv, qual, stored;
    } _Sft;
```

图 12-57（续）

```
        /* declarations */
FILE *_Foprep(const char *, const char *, FILE *);
int _Fopen(const char *, unsigned int, const char *);
int _Frprep(FILE *);
int _Ftmpnam(char *, int);
int _Fwprep(FILE *);
void _Genld(_Pft *, char, char *, short, short);
const char *_Getfld(_Sft *, const char *);
int _Getfloat(_Sft *);
int _Getint (_Sft *, char) ;
void _Ldtob(_Pft *, char);
void _Litob(_Pft *, char);
int _Printf(void *(*)(void *, const char *, size_t),
    void *, const char *, va_list);
void _Putfld(_Pft *, va_list *, char, char *);
int _Scanf(int (*)(void *, int),
    void *, const char *, va_list);
```

一旦 _Scanf 在一个格式中遇到百分号，它就开始分析后面的转换说明符。这个任务非常简单，因为扫描转换说明符的选择很少。对于几乎所有的转换说明符，_Scanf 也跳过前面的空白，只有少数几个除外。

函数 _Getfld

图 12-58 显示了文件 xgetfld.c，它定义了函数 _Getfld。_Scanf 调用这个函数来处理一个转换说明。这个函数由一个很大的 *switch* 语句组成，该 *switch* 语句分组处理转换说明符。_Getfld 从可变参数表中搜集需要的参数。（附属函数也搜集需要的参数。）它直接处理只包含文本的任何转换。

函数 _Getint

函数 _Getfld 通过调用函数 _Getint 执行所有的整数转换。图 12-59 显示了文件 xgetint.c，它定义了函数 _Getint。该函数搜集和一个整数值的适当模式匹配的所有字符，然后调用 <stdlib.h> 中声明的 strtol 或者 strtoul 来转换字段。头文件 "xstdio.h" 把宏 FMAX 定义为 512，这就有点超出了 C 标准要求的扫描函数必须转换的最长的字段长度。

指向 *void* 的指针

p 转换说明符和打印函数中这个相同的转换说明符含义相同。当然，C 标准中，扫描 *void* 指针的方式也是留给实现定义的。在这个实现中，我选择把字段作为一个 *unsigned long* 类型转换，然后把它存储为一个指向 *void* 的指针。我再次强调一下，这种做法不能保证在一个给定的体系结构下正确或者安全。在某些机器下，这里的代码可能必须经过修改才能有效地工作。

函数 _Getfloat

函数 _Getfld 通过调用 _Getfloat 执行所有的浮点转换。图 12-60 显示了文件 xgetfloat.c，这个文件定义了函数 _Getfloat。它搜集和一个浮点值的适当模式匹配的所有字符，然后调用 <stdlib.h> 中声明的 strtod 来转换字段。注意，即使是一个 *long double* 类型的存储值也要用 strtod 转换。如果 *long double* 的精度或者范围比 *double* 大，那么这样可能会限制能正确转换的值的范围。然而，这都是 C 标准要求的。当然，也可以使用一个可接受的扩展，编写一个“将字符串转换为 *long double* 类型”的函数（当然要使用私有名字）。这里我选择不承担这些额外的工作。

12

图 12-58

xgetfld.c

```
/* _Getfld function */
#include <ctype.h>
#include <limits.h>
#include <string.h>
#include "xstdio.h"

const char *_Getfld(_Sft *px, const char *s)
    {                                           /* convert a field */
    int ch;
    char *p;

    px->stored = 0;
    switch (*s)
        {                           /* switch on conversion specifier */
    case 'c':                           /* convert an array of chars */
        if (px->width == 0)
            px->width = 1;
        p = va_arg(px->ap, char *);
        for (; 0 < px->width; --px->width)
            if ((ch = GET(px)) < 0)
                return (NULL);
            else if (!px->noconv)
                *p++ = ch, px->stored = 1;
        break;
    case 'p':                                   /* convert a pointer */
    case 'd': case 'i': case 'o':
    case 'u': case 'x': case 'X':
        if (_Getint(px, *s))                   /* convert an integer */
            return (NULL);
        break;
    case 'e': case 'E': case 'f':
    case 'g': case 'G':
        if (_Getfloat(px))                      /* convert a floating
*/
            return (NULL);
        break;
    case 'n':                                /* return output count */
        if (px->quai == 'h')
            *va_arg(px->ap, short *) = px->nchar;
        else if (px->qual != 'l')
            *va_arg(px->ap, int *) = px->nchar;
        else
            *va_arg(px->ap, long *) = px->nchar;
        break;
    case 's':                                   /* convert a string */
        px->nget = px->width <= 0 ? INT_MAX : px->width;
        p = va_arg(px->ap, char *);
        while (0 <= (ch = GETN(px)))
            if (isspace(ch))
                break;
            else if (!px->noconv)
                *p++ = ch;
       UNGETN(px, ch);
       if(!px->noconv)
          *p++ = '\0', px->stored = 1;
       break;
```

图 12-58
（续）

```
	case '%':                                    /* match a '%' */
		if ((ch = GET(px)) == '%')
			break;
		UNGET(px,ch);
		return (NULL);
	case '[':
		{                                        /* convert a scan set */
		char comp = *++s == '^' ? *s++ : '\0';
		const char *t = strchr(*s == ']' ? s + 1 : s, ']');
		size_t n = t - s;

		if (t == NULL)
			return (NULL);                       /* undefined */
		px->nget = px->width <= 0 ? INT_MAX : px->width;
		p = va_arg(px->ap, char *);
		while (0 <= (ch = GETN(px)))
			if (!comp && !memchr(s, ch, n)
				|| comp && memchr(s, ch, n))
				break;
			else if (!px->noconv)
				*p++ = ch;
		UNGETN(px, ch);
		if (!px->noconv)
			*p++ = '\0', px->stored = 1;
		s = t;
		}
		break;
	default :                          /* undefined specifier, quit */
		return (NULL);
		}
	return (s);
	}
```

扫描函数的代码到这里就没有了，和打印函数一样，转换浮点值也要付出相当多的努力。扫描函数也涉及了很多代码。在一个小计算系统下，标准C 的实现对扫描浮点值的需求可能比打印它们要小。如果需要扫描函数但是不需要浮点支持，就可以通过提供 _Getfld 的修改的版本来大幅度缩减程序的大小。这样的方式也适用于本节前面讨论的打印函数。

12.5 <stdio.h> 的测试

头文件 <stdio.h> 声明的函数太多了而不能一次测试全部。（至少在本书中对 C 源文件大小的限制下不能。）我选择在一个测试程序中执行打印和扫描函数，另一个程序只测试所有的低级函数。

程序 tstdio1.c 图 12-61 显示了文件 tstdio1.c，它检查打印和扫描转换是否准确，如果它们的准确性不能保证就检查其精确程度。作为结果，它显示了几个宏的值。而且在把最后的输出行连接到一起时，它还执行了函数 vfprintf、vprintf 和 sprintf。对于这个实现，这个程序显示类似下面的输出并成功终止：

```
BUFSIZ = 512
L_tmpnam = 16
FILENAME_MAX = 64
FOPEN_MAX = 16
TMP_MAX = 32
SUCCESS testing <stdio. h>, part 1
```

12

图 12-59
xgetint.c

```
/* _Getint function */
#include <stdlib.h>
#include <string.h>
#include "xstdio.h"

int _Getint(_Sft *px, char code)
	{				/* get an integer value for _Scanf */
	char ac[FMAX+1], *p;
	char seendig = 0;
	int ch;
	static const char digits[]
		= "0123456789abcdefABCDEF";
	static const char flit[] = "diouxXp";
	static const char bases[] = {10, 0, 8, 10, 16, 16, 16};
	int base = bases[(const char *)strchr(flit, code) - flit];
	int dlen;

	px->nget = px->width <= 0
		|| FMAX < px->width ? FMAX : px->width;
	p = ac, ch = GETN(px);
	if (ch == '+' || ch == '-')
		*p++ = ch, ch = GETN(px);
	if (ch == '0')
		{				/* match possible prefix */
		seendig = 1;
		*p++ = ch, ch = GETN(px);
		if ((ch == 'x' || ch == 'X')
			&& (base == 0 || base == 16))
			base = 16, *p++ = ch, ch = GETN(px);
		else
			base = 8;
		}
	dlen =base == 0 || base == 10 ? 10 : base == 8 ? 8 : 16+6;
	for (; memchr(digits, ch, dlen); seendig = 1)
		*p++ = ch, ch = GETN(px);
	UNGETN(px, ch);
	if (!seendig)
		return (-1);
	*p = '\0';
	if (px->noconv)
		;
	else if (code == 'd' || code == 'i')
		{				/* deliver a signed integer */
		long lval = strtol(ac, NULL, base);

		px->stored = 1;
		if (px->qual == 'h')
			*va_arg(px->ap, short *) = lval;
		else if (px->qual != 'l')
			*va_arg(px->ap, int *) = lval;
		else
			*va_arg(px->ap, long *) = lval;
		}
```

图 12-59（续）

```
else
    {                           /* deliver an unsigned integer */
    unsigned long ulval = strtoul(ac, NULL, base);

    px->stored = 1;
    if (code == 'p')
        *va_arg(px->ap, void **) = (void *)ulval;
    else if (px->qual == 'h')
        *va_arg(px->ap, unsigned short *) = ulval;
    else if (px->qual != 'l')
        *va_arg(px->ap, unsigned int *) = ulval;
    else
        *va_arg(px->ap, unsigned long *) = ulval;
    }
return (0);
}
```

程序 tstdio2.c 图 12-62 显示了文件 tstdio2.c。它检查了这个头文件中定义的宏的属性，然后简单地执行了各种函数。一个输出信息来自对 perror 的调用（测试这个函数的时候一定要有输出——也可以制造所有可能的错误）。如果程序成功执行，输出以下内容：

```
Domain error reported as: domain error
SUCCESS testing <stdio.h>, part 2
```

12.6 参考文献

Brian W. Kernighan and P.J. Plauger, *Software Tools* (Reading, Mass.: Addison-Wesley, 1975). Also by the same authors, *Software Tools in Pascal* (Reading, Mass.: Addison-Wesley, 1978). 这些书都举例说明了怎样通过实现很少的基本接口函数来在各种操作系统下利用 UNIX I/O 模块。

William D. Clinger, "How to Read Floating-Point Numbers Accurately,"*Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation* (New York: Association for Computing Machinery, 1990, pp. 92-101). 这篇文章讨论了在保持全精度的前提下，把文本串转换为浮点表示的难点。

Guy L. Steele, Jr. and Jon L. White, "How to Print Floating-Point Numbers Accurately,"*Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation* (New York: Association for Computing Machinery, 1990, pp. 112-126). 这篇文章和上一篇文章出自同一个会议学报，它们刚好相互对应。

12.7 习题

12.1 你使用的操作系统怎样表示文本文件？需要作一些改动来和标准 C 中文本流的内部表示相匹配吗？

12

12.2 调用函数 vfprintf 和 vsprintf 来编写函数 fprintf、printf 和 sprintf。

12.3 编写函数 rename 的一个版本，当它不能简单地对文件重命名时，就复制这个文件。成功地复制文件之后要删除原始文件。

12.4 编写函数 remove 的一个版本，使它只是对要删除的文件重命名。把文件放在一个不常用的目录下，或者用一个不会和文件的普通命名习惯冲突的名字对它进行命名。这种功能有什么好处？

12.5 编写函数 tmpnam 的一个版本，它要检查是否和已存的名字冲突（试着用那个文件名打开一个已存的文件以便读取）。这个函数可以一直生成新的文件名，直到它不能打开相应的文件。这种功能有什么好处？如果两个并行执行的程序同时调用这个函数会发生什么？

C 标准中讲到，“实现的行为不受调用函数 tmpnam 的影响”（参考 12.2 节），怎样做才能满足这个要求？

12.6 为你使用的操作系统实现原语 _Fclose、_Fopen、_Fread 和 _Fwrite。需要使用汇编语言吗？

12.7 [**难**] 为一个操作系统实现函数 _Fgetpos 和 _Fsetpos，该系统使用回车加换行来终止每个文本行。

12.8 [**难**] 编写一个函数，它把文本串转换为 *long double* 类型，遵循的规则和 strtod 对 *double* 类型使用的规则（参考 13.4 节）相同。

12.9 [**很难**] 重新设计扫描函数，使它们的用途更广。

设计一种能使扫描错误和调用程序通信的方法，使得它能实现以下几个功能：

- 更精确地确定错误点；
- 尝试另一种转换；
- 从一个读错误中很好地恢复。

图 12-60 xgetfloat.c

```
/* _Getfloat function */
#include <ctype.h>
#include <locale.h>
#include <stdlib.h>
#include <string.h>
#include "xstdio.h"

int _Getfloat (_Sft *px)
    {                       /* get a floating point value for _scanf */
    char *p;
    int ch;
    char ac[FMAX+1];
    char seendig = 0;

    px->nget = px->width <= 0
        || FMAX < px->width ? FMAX : px->width;
    p = ac, ch = GETN(px);
    if (ch == '+' || ch == '-')
        *p++ = ch, ch = GETN(px);
    for (; isdigit(ch); seendig = 1)
        *p++ = ch, ch = GETN(px);
    if (ch == localeconv()->decimal_point[0])
        *p++ = ch, ch = GETN(px);
    for (; isdigit(ch); seendig = 1)
        *p++ = ch, ch = GETN(px);
    if ((ch == 'e' || ch == 'E' ) && seendig)
        {                                       /* parse exponent */
        *p++ = ch, ch = GETN(px);
        if (ch == '+' || ch == '-')
            *p++ = ch, ch = GETN(px);
        for (seendig = 0; isdigit(ch); seendig = 1)
            *p++ = ch, ch = GETN(px);
        }
    UNGETN(px, ch);
    if (! seendig)
        return (-1);
    *p = '\0';
    if (!px->noconv)
        {                                   /* convert and store */
        double dval = strtod(ac, NULL);

        px->stored = 1;
        if (px->qual == 'l')
            *va_arg(px->ap, double *) = dval;
        else if (px->qual != 'L')
            *va_arg(px->ap, float *) = dval;
        else
            *va_arg(px->ap, long double *) = dval;
        }
    return (0);
    }
```
□

12

图 12-61 tstdio1.c

```
/* test stdio functions, part 1 */
#include <assert.h>
#include <errno.h>
#include <float.h>
#include <math.h
#include <stdarg.h>
#include <stdio.h>
#include <string.h>

static void vfp(const char *fmt, ...)
    {                                           /* test vfprintf */
    va_list ap;

    va_start(ap, fmt);
    vfprintf(stdout, fmt, ap);
    va_end(ap);
    }

static void vp(const char *fmt, ...)
    {                                            /* test vprintf */
    va_list ap;

    va_start(ap, fmt);
    vprintf(fmt, ap);
    va_end(ap);
    }

static void vsp(char *s, const char *fmt, ...)
    {                                           /* test vsprintf */
    va_list ap;

    va_start(ap, fmt);
    vsprintf(s, fmt, ap);
    va_end(ap);
    }

int main()
    {                     /* test basic workings of stdio functions */
    char buf[32], ch;
    double db;
    float fl;
    int in;
    long lo;
    long double ld;
    short sh;
    void *pv;

    assert(sprintf(buf, "%2c|%-4d|%.4o|%#lX",
        'a', -4, 8, 12L) == 16);
    assert(strcmp(buf, " a|-4 |0010|OXC") == 0);
    assert(sscanf (buf, " %c| % hd | %i |%l x",
        &ch, &sh, &in, &lo) == 4);
    assert(ch == 'a' && sh == -4 && in == 8 && lo == 12);
    assert(sprintf(buf, "%E|%.2f|%Lg",
        1.1e20, -3.346, .02L) == 23);
```

图 12-61
（续）

```
assert(strcmp(buf, "1.100000E+20|-3.35|0.02") == 0);
assert(sscanf(buf, "%e|%lg|%Lf", &fl, &db, &ld) == 3);
assert(fabs(fl - 1.1e20) / 1.1e20 < 4 * FLT_EPSILON);
assert(fabs(db + 3.35) / 3.35 < 4 * DBL_EPSILON);
assert(fabs(ld - 0.02) / 0.02 < 4 * LDBL_EPSILON);
assert(4 <= sprintf(buf, "|%%%n %p",
    &in, (void *)&ch) && in == 2);
assert(sscanf(buf, "|%%%n %p", &in, &pv) == 1 && in == 2);
 {                                       /* test formatted I/O */
char buf[10];
const char *tn = tmpnam(NULL);
FILE *pf;
fpos_t fp1, fp2;
int in1, in2;
long off;

assert(tn != NULL && (pf = fopen(tn, "w+")) != NULL);
setbuf(pf, NULL);
assert(fprintf(pf, "123\n") == 4);
assert((off = ftell(pf)) != -1);
assert(fprintf(pf, "456\n") == 4);
assert(fgetpos(pf, &fp1) == 0);
assert(fprintf(pf, "789\n") == 4);
rewind(pf);
assert(fscanf(pf, "%i", &in1) == 1 && in1 == 123);
assert(fsetpos(pf, &fp1) == 0);
assert(fscanf(pf, "%i", &in1) == 1 && in1 == 789);
assert(fseek(pf, off, SEEK_SET) == 0);
assert(fscanf(pf, "%i", &in1) == 1 && in1 == 456);
assert(fclose(pf) == 0
    && freopen(tn, "r", stdin) == stdin);
assert(setvbuf(stdin, buf, _IOLBF, sizeof (buf)) == 0);
assert(scanf("%i", &in1) == 1 && in1 == 123);
assert(fclose(stdin) == 0);
assert((pf = fopen(tn, "w+b")) != NULL);
 }
printf("BUFSIZ = %u\n", BUFSIZ) ;
printf("L_tmpnam = %u\n", L_tmpnam);
printf("FILENAME_MAX = %u\n", FILENAME_MAX);
```

12

图 12-62
tstdio2.c

```c
/* test stdio functions, part 2 */
#include <assert.h>
#include <errno.h>
#include <stdio.h>
#include <string.h>

int main()
    {                   /* test basic workings of stdio functions */
    char buf[32], tname[L_tmpnam], *tn;
    FILE *pf;
    static int macs[] = {
        _IOFBF, _IOLBF, _IONBF, BUFSIZ, EOF, FILENAME_MAX,
        FOPEN_MAX, TMP_MAX, SEEK_CUR, SEEK_END, SEEK_SET};

    assert(256 <= BUFSIZ && EOF < 0);
    assert(8 <= FOPEN_MAX && 25 <= TMP_MAX);
    assert(tmpnam(tname) == tname && strlen(tname) < L_tmpnam);
    assert((tn = tmpnam(NULL)) ! = NULL
        && strcmp(tn, tname) != 0);
    pf = fopen(tname, "w");
    assert(pf != NULL
        && pf != stdin && pf != stdout && pf != stderr);
    assert(feof(pf) == 0 && ferror(pf) == 0);
    assert(fgetc(pf) == EOF
        && feof(pf) == 0 && ferror(pf) != 0);
    clearerr(pf);
    assert(ferror(pf) == 0);
    assert(fputc('a', pf) == 'a' && putc('b', pf) == 'b');
    assert(0 <= fputs("cde\n", pf));
    assert(0 <= fputs("fghij\n", pf));
    assert(fflush(pf) == 0);
    assert(fwrite("klmnopq\n", 2, 4, pf) == 4);
    assert(fclose(pf) == 0);
    assert(freopen(tname, "r", stdin) == stdin);
    assert(fgetc(stdin) == 'a' && getc(stdin) == 'b');
    assert(getchar() == 'c');
    assert(fgets(buf, sizeof (buf), stdin) == buf
        && strcmp(buf, "de\n") == 0);
    assert(ungetc('x', stdin) == 'x');
    assert(gets(buf) == buf && strcmp(buf, "xfghij") == 0);
    assert(fread(buf, 2, 4, stdin) == 4
        && strncmp(buf, "klmnopq\n", 8, ) == 0);
    assert(getchar() == EOF && feof(stdin) != 0);
    remove(tn);
    assert(rename (tname, tn) == 0
        && fopen(tname, "r") == NULL);
    assert((pf = fopen (tn, "r")) != NULL && fclose(pf) == 0);
    assert(remove(tn) == 0 && fopen(tn, "r") == NULL);
    assert((pf = tmpfile()) != NULL && fputc('x', pf) == 'x');
    errno = EDOM;
    perror("Domain error reported as");
    putchar('S'), puts("UCCESS testing <stdio.h>, part 2");
    return(0);
    }
```

第 13 章

<stdlib.h>

13.1 背景知识

头文件 <stdlib.h> 是一个大杂烩，X3J11 委员会发明这个头文件是为了定义和声明那些没有明显的归属的宏和函数。

- 很多存在的函数，例如 abs 和 malloc，传统的头文件没有声明它们。X3J11 强烈认为每个函数都应该在一个标准头文件声明。如果所有其他的头文件都没有声明这样一个函数，那么 <stdlib.h> 就会声明它。
- 新的宏和函数组可以放在新的标准头文件中，<float.h> 和 <locale.h> 就是明显的例子。对已经存在的分组的补充放在已经存在的头文件中，例如 <string.h> 中声明的 strcoll 和 <time.h> 中声明的 strftime。还有其他一些宏和函数难于分类，那么它们就在 <stdlib.h> 中定义或者声明。

这个头文件不仅仅是一个大杂烩。11.1 节中讨论了头文件 <stddef.h> 的演化。

函数分组 为了使这一章结构更合理，我把这些函数分为 6 组：

- 整型数学（abs、div、labs 和 ldiv）——执行简单的整型算术；
- 算法（bsearch、qsort、rand 和 srand）——收集那些复杂而又被广泛使用的、足以打包作为库函数的操作；
- 文本转换（atof、atoi、atol、strtod、strtol 和 strtoul）——确定文本表示的编码算术值；
- 多字节转换（mblen、mbstowcs、mbtowc、wcstombs 和 wctomb）——多字节和宽字节字符编码之间的转换；
- 存储分配（calloc、free、malloc 和 realloc）——管理数据对象的堆；
- 环境接口（abort、atexit、exit、getenv 和 system）——程序和执行环境之间的接口。

对每一组我都单独地讨论怎样实现其中的函数。

13.2 C 标准的内容

<stdlib.h>

7.10 一般功能 <stdlib.h>

头文件 <stdlib.h> 声明了 4 种类型和几个具有一般功能的函数，还定义了几个宏[126]。

size_t wchar_t div_t

声明的类型有 size_t 和 wchar_t（都在 7.1.6 中描述），

div_t

它是一个结构类型，也是函数 div 的返回值的类型，

ldiv_t

ldiv_t

也是一个结构类型，是函数 ldiv 的返回值类型。

NULL EXIT_FAILURE

定义的宏有 NULL（7.1.6 中描述），

EXIT_FAILURE

和

EXIT_SUCCESS

EXIT_SUCCESS

它们展开为整数表达式，可以作为 exit 函数的参数使用，分别返回给宿主环境不成功和成功终止的状态。

RAND_MAX

RAND_MAX

它展开为整值常量表达式，其值是 rand 函数返回的最大值。

MB_CUR_MAX

MB_CUR_MAX

它展开为正整数表达式，其值是当前区域设置（类型 LC_CTYPE）指定的扩展字符集中多字节字符的最大字节数目，并且绝不会比 MB_LEN_MAX 大。

7.10.1 字符转换函数

函数 atof、atoi 和 atol 不会影响错误的整数表达式 errno 的值。如果结果的值不能表示，则行为是未定义的。

atof

7.10.1.1 函数 atof

概述

```
#include <stdlib.h>
double atof(const char *nptr);
```

说明

函数 atof 把 nprt 指向的字符串的初始部分转换为 double 类型的表示。除了出错后的行为，它等价于：

```
strtod(nptr, (char **)NULL)
```

返回值

函数 atof 返回转换后的值。

参见：函数 strtod（7.10.1.4）。

atoi

7.10.1.2 函数 atoi

概述

```
#include <stdlib.h>
int atoi(const char *nptr);
```

说明

函数 atoi 把 nprt 指向的字符串的初始部分转换为 int 表示。除了出错后的行为，它等价于：

```
(int)strtol(nptr,(char **)NULL,10)
```

返回值

函数 atoi 返回转换后的值。

参见：函数 strtol（7.10.1.5）。

atol

7.10.1.3 函数 atol

概述

```
#include <stdlib.h>
long int atol(const char *nptr);
```

说明

函数 atol 把 nprt 指向的字符串的初始部分转换为 long int 表示。除了出错后的行为，它等价于：

```
strtol(nptr,(char **)NULL,10)
```

返回值

函数 atol 返回转换后的值。

参见：函数 strtol（7.10.1.5）。

strtod

7.10.1.4 函数 strtod

概述

```
#include <stdlib.h>
double strtod(const char *nptr,char **endptr);
```

说明

函数 strtod 把 nptr 指向的串的初始部分转换为 double 表示。首先，它把输入串分解为 3 个部分：可能为空的初始序列，由一些空白字符（由 isspace 函数指定）组成；目标序列浮点常量相似的序列；最后一个序列由一个或者多个不能识别的字符组成，包括输入串的结束的空字符。然后，它试着把目标序列转换为一个浮点数并返回结果。

目标序列的期望形式是一个可选的加号或者减号，然后是非空的数字序列，其中可能包括小数点，接着是可选的指数部分，如 6.1.3.1 中所定义的，但没有浮点后缀。目标序列定义为输入串的最长的初始序列，以第一个非空白字符开头，这也是期望的形式。如果输入串为空或者完全由空白组成，或者第一个非空白字符不是符号、数字或者小数点字符，则目标序列不包含任何字符。

如果目标序列是期望的形式，字符序列以一个数字或者小数点字符开头（不论哪一个在前），它就会根据 6.1.3.1 的规则被解释为一个浮点常量，除非小数点是用来代替句号，或者假定小数点紧跟着串的最后一个数字(如果既没有指数部分也没有小数点字符出现)如果中间序列以一个减号开头，则转换后的值就变为它的相反数。如果 endptr 不为空，则指向最后串的指针存储在 endptr 指向的对象中。

在 "C" 区域设置之外，实现定义的其他目标序列形式也能被接受。

如果目标序列为空或者不是期望的形式，则不执行任何转换。如果 endptr 不为空，则 nptr 的值就存储在 endptr 指向的数据对象中。

返回值

如果存在转换后的值，函数 strtod 就返回这个值。如果没有执行转换，则返回零。如果正确的转换值在可表示值的范围外，则返回正或者负(根据值的符号)的 HUGE_VAL，并将宏 ERANGE 的值存储在 errno 中。如果正确的转换值会造成下溢，则返回零，并将宏 ERANGE 的值存储在 errno 中。

strtol

7.10.1.5 函数 strtol

概述

```
#include <stdlib.h>
long int strtol(const char *nptr,char **endptr,int base);
```

说明

函数 strtol 把 nptr 指向的串的初始部分转换为 long int 表示。首先，它把输入串分解为 3 个部分：可能为空的初始序列，由空白字符（由 isspace 函数指定）组成；目标序列是一个和基数为 base 的整数表示相似的序列；最后一个序列由一个或者多个不能识别的字

符组成，包括输入串的结束的空字符。然后，它试着把目标序列转换为一个整数并返回结果。

如果 base 的值为零，目标序列的期望形式就是 6.1.3.2 描述的整数常量，前面可以有一个加号或者减号，但不包括整数后缀。如果 base 的值在 2 到 36 之间，那么目标序列的期望形式就是字母和数字的序列，它表示以 base 为基数的整数，前面可以有一个加号或者减号，但不包括整数后缀。字母 a（或者 A）到 z（或者 Z）分别代表值 10 到 35，只允许使用代表的值小于 base 的那些字母。如果 base 的值是 16，字符 0x 或者 0X 也可以放在含有字母和数字的序列之前，如果有符号的话，就紧跟符号。

目标序列被定义为输入串的最长的初始序列，以第一个非空白字符处开始，这就是期望的形式。如果输入串为空或者完全由空白组成，或者第一个非空白字符不是符号或允许使用的字母或数字，那么中间序列不包含任何字符。

如果目标序列是期望的形式，而且 base 的值为零，以第一个数字开头的字符序列就会根据 6.1.3.2 的规则被解释为一个整数常量。如果目标序列是期望的形式，而且 base 的值在 2 和 36 之间，它就被用作转换的基数，和它对应的每一个字母如上所示。如果目标序列以一个减号开头，转换后的值就变为它的相反数。如果 endptr 不是一个空指针，那么指向最后串的指针就存储在 endptr 指向的数据对象中。

在 "C" 区域设置之外，实现定义的其他的中间序列形式也能被接受。

如果目标序列为空或者不是期望的形式，则不执行任何转换。如果 endptr 不为空，则 nptr 的值就存储在 endptr 指向的对象中。

返回值

如果存在转换后的值，函数 strtol 就返回这个值。如果没有执行转换，则返回零。如果正确的转换值在可表示值的范围外，则（根据值的符号）返回 LONG_MAX 或者 LONG_MIN，并把宏 ERANGE 存储在 errno 中。

strtoul

7.10.1.6 函数 strtoul

概述

```
#include <stdlib.h>
unsigned long int strtoul(const char *nptr,char **endptr,int base);
```

说明

函数 strtoul 把 nptr 指向的串的初始部分转换为 unsigned long int 表示。首先，它把输入串分解为 3 个部分：可能为空的初始序列，由空白字符（由 isspace 函数指定）组成；目标序列是一个和基数为 base 的无符号整数表示相似的序列；最后一个序列由一个或者多个不能识别的字符组成，包括输入串的结束的空字符。然后，它试着把目标序列转换为一个无符号整数并返回结果。

如果 base 的值为零，目标序列的期望形式就是 6.1.3.2 描述的整数常量，前面可以有一个加号或者减号，但不包括整数后缀。如果 base 的值在 2 到 36 之间，那么目标序列的期望形式就是字母和数字的序列，它表示以 base 为基数的整数，前面可以有一个加号或者减号，但不包括整数后缀。字母 a（或者 A）到 z（或者 Z）分别代表值 10 到 35，只允许使用代表的值小于 base 的那些字母。如果 base 的值是 16，字符 0x 或者 0X 也可以放在字母和数字的序列之前，如果有符号的话，就紧跟符号。

目标序列被定义为输入串的最长的初始序列，以第一个非空白字符处开始，这就是期望的形式。如果输入串为空或者完全由空白组成，或者第一个非空白字符不是符号或允许使用的字母或数字，那么目标序列不包含任何字符。

如果目标序列是期望的形式，而且 base 的值为零，以第一个数字开头的字符序列就会根据 6.1.3.2 的规则被解释为一个整数常量。如果目标序列是期望的形式，而且 base 的值在 2 和 36 之间，它就是转换的基数，和它对应的每一个字母如上所示。如果目标序列

以一个减号开头，转换后的值就变为它的相反数。如果 endptr 不是一个空指针，那么指向最后串的指针就存储在 endptr 指向的数据对象中。

在 "C" 区域设置之外，实现定义的其他的中间序列形式也能被接受。

如果目标序列为空或者不是期望的形式，则不执行任何转换。如果 endptr 不为空，则 nptr 的值就存储在 endptr 指向的对象中。

返回值

如果存在转换后的值，函数 strtoul 就返回这个值。如果没有执行转换，则返回零。如果正确的转换值在可表示值的范围外，则返回 ULONG_MAX，并把宏 ERANGE 存储在 errno 中。

7.10.2 伪随机序列产生函数

rand

7.10.2.1 函数 rand

概述

```
#include <stdlib.h>
int rand(void);
```

说明

函数 rand 计算一个伪随机整数序列，它们的范围在 0 到 RAND_MAX 之间。

实现的行为不受 rand 函数的影响。

返回值

函数 rand 返回一个伪随机整数。

环境限制

宏 RAND_MAX 的值至少应该为 32 767。

srand

7.10.2.2 函数 srand

概述

```
#include <stdlib.h>
void srand(unsigned int seed);
```

说明

函数 srand 使用参数作为一个新的伪随机数序列的种子，这个序列由后来对 rand 的调用返回。如果再用相同的种子调用 srand，那么伪随机数序列就会重复。如果在对 srand 的任何调用之前调用 rand，那么它产生的序列和用 1 为参数第一次调用 srand 之后生成的序列相同。

实现的行为不受 srand 函数调用的影响。

返回值

函数 srand 没有返回值。

例子

下面的函数定义了 rand 和 srand 的一个可移植的实现。

```
static unsigned long int next = 1;

int rand(void) /* RAND_MAX assumed to be 32767 */
{
     next = next * 1103515245 + 12345;
     return (unsigned int) (next/65536) % 32768;
}

void srand(unsigned int seed)
{
      next = seed;
}
```

13

7.10.3 内存管理函数

通过连续调用函数 calloc、malloc 和 realloc 所分配的空间的顺序和连续性是不确定的。如果分配成功，那么返回的指针就能被适当赋值为指向任意类型的对象的指针，然后就可以用它来访问存储在所分配的空间中的这种数据对象或者这种数据对象的数组（直到空间被显式地释放或者重新分配）。每一次分配都应该产生一个不和其他对象相连的指针，返回的指针指向分配空间的开始位置（最低字节地址）。如果不能分配空间，就返回空指针。如果要求分配的空间大小是零，那么行为是由实现定义的，返回的值是一个空指针或者一个特殊的指针。空间释放后指针的值是不确定的。

calloc

7.10.3.1 函数 calloc

概述

```
#include <stdlib.h>
void *calloc(size_t nmemb, size_t size);
```

说明

函数 calloc 为 nmemb 个对象的数组分配空间，每一个元素的大小为 size，分配空间的所有位都被初始化为零[127]。

返回值

函数 calloc 返回一个空指针或者指向分配的空间的指针。

free

7.10.3.2 函数 free

概述

```
#include <stdlib.h>
void free(void *ptr);
```

说明

函数 free 释放 ptr 指向的空间，也就是说这部分空间以后还可以被分配。如果 ptr 是一个空指针，则不发生任何行为。否则，如果参数与 calloc、malloc 或者 realloc 之前返回的指针不匹配，或者它指向的空间已经被 free 和 realloc 释放了，那么行为是未定义的。

返回值

函数 free 没有返回值。

malloc

7.10.3.3 函数 malloc

概述

```
#include <stdlib.h>
void *malloc(size_t size);
```

说明

函数 malloc 为一个对象分配空间，其中，该对象的大小由size 指定且对象的值是不确定的。

返回值

函数 malloc 返回一个空指针或者指向分配空间的指针。

realloc

7.10.3.4 函数 realloc

概述

```
#include <stdlib.h>
void *realloc(void *ptr, size_t size);
```

说明

函数 realloc 将 ptr 指向的对象的大小改变为由 size 指定的大小。新旧空间中较小的一个中的对象内容应该保持不变。如果新空间比较大，那么对象的新分配部分的值是不确定的。如果 ptr 是一个空指针，那么对于一个指定的大小函数 realloc 和行为跟 malloc 相同；否则，如果 ptr 与 calloc、malloc 或者 realloc 之前返回的指针不匹配，或者它指向的空间已经被 free 或者 realloc 释放了，那么行为是未定义的。如果不能分配空间，则 ptr 指向的对象不变。如果 size 是零且 ptr 不是空指针，那么它指向的对象就会被释放。

返回值

函数 realloc 返回空指针或者指向可能被移动了的分配空间的指针。

7.10.4 环境通信函数

abort

7.10.4.1 函数 abort

概述

```
#include <stdlib.h>
void abort (void);
```

说明

函数 abort 使程序异常终止，除非捕获了信号 SIGABRT 且信号处理程序没有返回。是清空打开的输出流，还是关闭打开的流还是移除临时文件是由实现定义的。一种由实现定义的不成功的终止状态会通过函数调用 raise (SIGABRT) 返回给宿主环境。

返回值

函数 abort 不能返回到它的调用者。

atexit

7.10.4.2 函数 atexit

概述

```
#include <stdlib.h>
int atexit (void (*func) (void) );
```

说明

函数 atexit 注册 func 指向的函数，该函数在程序正常终止的时候被调用，而不需要任何参数。

实现限制

实现应该支持至少 32 个函数的注册。

返回值

如果注册成功，函数 atexit 返回零，否则返回非零。

参见： 函数 exit (7.10.4.3)。

exit

7.10.4.3 函数 exit

概述

```
#include <stdlib.h>
void exit (int status);
```

说明

函数 exit 使程序的执行正常终止。如果程序执行了对 exit 的多次调用，则这种行为未定义。

首先，调用所有 atexit 注册的函数，按照它们注册时的反顺序调用[128]。

然后，清空所有打开的具有未写缓冲数据的流，关闭所有打开的流，并删除 tmpfile 函数创建的所有文件。

最后，控制权返回给宿主环境。如果 status 的值是零或者 EXIT_SUCCESS，则返回一种由实现定义的成功终止状态。如果 status 的值是 EXIT_FAILURE，则返回一种由实现定义的不成功终止状态。否则，返回的状态是由实现定义的。

返回值

函数 exit 不能返回到它的调用者。

getenv

7.10.4.4 函数 getenv

概述

```
#include <stdlib.h>
char *getenv(const char *name);
```

13

说明

函数 getenv 搜索宿主环境提供的环境表，来查找一个能和 name 指向的串匹配的串。环境名字集和修改环境表的方法是由实现定义的。

实现的行为不受 getenv 函数调用的影响。

返回值

函数 getenv 返回一个指向串的指针，这个串与匹配的列表成员相关。指向的串不能被程序修改，但是可以通过后续调用函数 getenv 被覆盖。如果不能找到指定的 name，则返回空指针。

system

7.10.4.5 函数 system

概述

```
#include <stdlib.h>
int system(const char *string);
```

说明

函数 system 把 string 指向的串传递给宿主环境，然后命令处理程序按照实现定义的方式执行它。可以使用空指针作为参数查询命令处理程序是否存在。

返回值

如果参数是一个空指针，那么只有当命令处理程序可用的时候函数 system 返回非零。如果参数不是空指针，函数 system 返回一个实现定义的值。

7.10.5 查找和排序函数

bsearch

7.10.5.1 函数 bsearch

概述

```
#include <stdlib.h>
void *bsearch(const void *key, const void *base, size_t nmemb,
      size_t size, int (*compar)(const void *, const void *));
```

说明

函数 bsearch 搜索一个 nmemb 个对象的数组，来查找与 key 指向的对象匹配的元素，base 指向这个数组的第一个元素。数组中每一个元素的大小由 size 指定。

用两个参数调用 compar 指向的比较函数，第 1 个参数指向 key 数据对象第 2 个参数指向一个数组元素。如果 key 对象小于、等于或者大于数组元素，则函数将分别返回一个小于零、等于零或者大于零的整数。此时，数组的前一部分存储的应该是小于 key 的元素，中间一部分是等于 key 的元素，而最后一部分是大于 key 的元素[129]。

返回值

函数 bsearch 返回指向数组中与 key 匹配的元素的指针，如果找不到这样的元素，则返回空指针。如果两个元素比较时相等，则未指定和哪个元素匹配。

qsort

7.10.5.2 函数 qsort

概述

```
#include <stdlib.h>
void qsort(void *base, size_t nmemb, size_t size,
      int (*compar)(const void *, const void *));
```

说明

函数 qsort 对一个包含 nmemb 个对象的数组进行排序，其第一个元素由 base 指向，数组中每个元素的大小是由 size 指定。

根据 compar 所指向的比较函数将数组内容排列成升序。函数的参数分别指向两个数组元素的指针。如果第一个元素小于、等于或者大于第二个元素，则函数分别返回小于零、等于零和大于零的整数。

如果比较的两个元素相等，则没有指定它们在排列后的数组中的顺序。

返回值

函数 qsort 没有返回值。

7.10.6 整数算术函数

abs

7.10.6.1 函数 abs

概述

```
#include <stdlib.h>
int abs(int j);
```

说明

函数 abs 计算整数 j 的绝对值。如果结果不能表示，则这种行为未定义[130]。

返回值

函数 abs 返回绝对值。

div

7.10.6.2 函数 div

概述

```
#include <stdlib.h>
div_t div(int numer, int denom);
```

说明

函数 div 计算分子 numer 除以分母 denom 所得的商和余数。如果不能整除，那么所得的商是小于它的代数商的最接近的整数。如果结果不能表示，那么行为未定义；否则 numer 等于 quot*denom+rem。

返回值

函数 div 返回一个 div_t 类型的结构，该结构由商和余数组成。结构应该包含以下成员，不分先后顺序：

```
int quot; /* quotient */
int rem; /* remainder */
```

labs

7.10.6.3 函数 labs

概述

```
#include <stdlib.h>
long int labs(long int j);
```

说明

函数 labs 和函数 abs 很相似，只不过它的参数和返回值的类型都是 long int。

ldiv

7.10.6.4 函数 ldiv

概述

```
#include <stdlib.h>
ldiv_t ldiv(long int numer, long int denom);
```

说明

函数 ldiv 和函数 div 很相似，只不过它的参数和返回的结构（ldiv_t 类型）的成员的类型都是 long int。

7.10.7 多字节字符函数

多字节字符函数的行为受当前区域设置的类别 LC_CTYPE 的影响。对一个状态相关的编码方式，每一个函数都通过调用它的字符指针参数 s，进入到这个函数的初始状态，s 是一个空指针。s 为非空指针时，使用 s 对函数的后续调用可以在必要的时候改变函数的内部状态。s 为空指针时，如果编码方式是状态相关的，那么这些函数就会返回一个非零值；否则返回零[131]。改变类别 LC_CTYPE 会导致这些函数的转移状态变得不确定。

13

mblen

7.10.7.1 函数 mblen

概述

```
#include <stdlib.h>
int mblen(const char *s, size_t n);
```

说明

如果 s 不是一个空指针，函数 mblen 就确定 s 指向的多字节字符中包含的字节数。除了 mbtowc 函数的转移状态不受影响，这个函数等价于：

```
mbtowc( (wchar_t *) 0, s, n) ;
```

实现的行为不受调用函数 mblen 的影响。

返回值

如果 s 是一个空指针，多字节字符编码是状态相关的或不是状态相关的时，函数 mblen 分别返回一个非零或者零值。如果 s 不是空指针，函数 mblen 或者返回零（如果 s 指向空字符），或者返回多字节字符包含的字节数（如果后面 n 个或者更少的字节形成一个有效的多字节字符），或者返回 -1（如果它们不能形成有效的多字节字符）。

参见：函数 mbtowc（7.10.7.2）。

mbtowc

7.10.7.2 函数 mbtowc

概述

```
#include <stdlib.h>
int mbtowc(wchar_t *pwc, const char *s, size_t n);
```

说明

如果 s 不是一个空指针，函数 mbtowc 确定 s 指向的多字节字符中包含的字节数，然后确定和这个多字节字符对应的 wchar_t 类型的值的编码。（和空字符对应的编码值是零。）如果多字节字符有效且 pwc 不是空指针，mbtowc 函数就把编码存储在 pwc 指向的对象中。最多可以检测 s 指向的数组中的 n 个字节。

实现的行为不受函数 mbtowc 调用的影响。

返回值

如果 s 是一个空指针，多字节字符编码是状态相关的或不是状态相关的时，函数 mbtowc 分别返回一个非零或者零值。如果 s 不是空指针，则函数 mbtowc 或者返回零（如果 s 指向空字符），或者返回转换的多字节字符中包含的字节数（如果后面 n 个或者更少的字节形成一个有效的多字节字符），或者返回 -1（如果它们不能形成一个有效的多字节字符）。

任何情况下返回的值都不能比 n 或者宏 MB_CUR_MAX 的值大。

wctomb

7.10.7.3 函数 wctomb

概述

```
#include <stdlib.h>
int wctomb(char *s, wchar_t wchar);
```

说明

函数 wctomb 确定表示多字节字符所需要的字节数，这个多字节字符和值为 wchar（包括转移状态中的任何改变）的编码相对应。它把多字节字符表示存储在 s（如果 s 不是空指针）指向的数组对象中，最多可以存储 MB_CUR_MAX 个字符。如果 wchar 的值是零，则函数仍为初始的转移状态。

实现的行为不受调用函数 wctomb 的影响。

返回值

如果 s 是一个空指针，多字节字符编码是状态相关的或不是状态相关的时，函数 wctomb 分别返回一个非零或者零值。如果 s 不是空指针，wchar 的值不能对应一个有效的多字节字符时，函数返回 -1；否则函数返回和 wchar 的值对应的多字节字符中包含的字节数。

任何情况下的返回值都不能比宏 MB_CUR_MAX 的值大。

7.10.8 多字节串函数

多字节串函数的行为受当前区域设置类别 LC_CTYPE 的影响。

mbstowcs

7.10.8.1 函数 mbstowcs

概述

```
#include <stdlib.h>
size_t mbstowcs(wchar_t *pwcs, const char *s, size_t n);
```

说明

函数 mbstowcs 将 s 指向的数组中以初始的转移状态开始的多字节字符序列转换为相应的编码序列，并将最多 n 个编码存放到 pwcs 指向的数组中。空字符（转换成值为零的编码）后面的多字节字符不会被检测或者转换。每一个多字节字符都像调用 mbtowc 一样进行转换，只不过函数 mbtowc 的转移状态不受影响。

pwcs 指向的数组中最多只能修改 n 个元素。如果复制发生在两个重叠的对象之间，那么行为未定义。

返回值

如果遇到无效的多字节字符，函数 mbstowcs 返回 (size_t) -1；否则，函数 mbstowcs 返回修改的数组元素的个数，不包括终止的零编码（如果有的话）[132]。

wcstombs

7.10.8.2 函数 wcstombs

概述

```
#include <stdlib.h>
size_t wcstombs(char *s, const wchar_t *pwcs, size_t n);
```

说明

函数 wcstombs 将以 pwcs 指向的数组中的多字节字符对应的编码序列转换成以初始的转移状态开始的多字节字符序列，并将结果存放在 s 指向的字符数组中。如果一个多字节字符超过了 n 个字节的限制或存储了一个空字符，则存储结束。每个编码像调用 wctomb 一样进行转换，只不过 wctomb 的转移状态不受影响。

由 s 指向的数组中最多只能修改 n 个字节。如果复制发生在两个重叠的对象之间，那么这种行为未定义。

返回值

如果遇到一个不能和有效的多字节字符对应的编码，那么函数 wcstombs 返回 (size_t)-1；否则，函数 wcstombs 返回修改的字节数，不包括终止的空字符（如果有的话）[133]。

脚注

126. 参考“库的展望”(7.1 3.7)。
127. 注意它不需要和浮点零或者空指针常量的表示相同。
128. 每个函数的调用次数与它的注册次数相同
129. 实际上，整个数组是根据比较函数来排序的。
130. 最小的负数的绝对值不能用 2 的补码表示。
131. 如果实现使用一些特殊的字节来改变转移状态，这些字节不会产生单独的宽字节字符编码，而是要跟临近的多字节字符编为一组。
132. 如果返回值是 n，那么数组不会是以空或者零结束。

13.3 <stdlib.h> 的使用

<stdlib.h> 中声明的很多函数都可以单独列出来。也许，atexit 要和 exit 一起使用，srand 要和 rand 一起使用。但仍然可以单独地使用和理解大部分函数。在这么多函数中，有两组更为重要：

- 存储分配函数一起工作来管理堆；
- 多字节函数一起工作来使大字符集合的不同表示之间可以相互转换。

这些组中的每一组函数都值得讨论。

存储分配函数

一个 C 标准程序的数据对象占据以下 3 种类型的存储空间：

- 程序在开始执行前分配*静态存储空间*并进行初始化。如果没有指定一个数据对象（部分或者全部）的初始值，程序就会把它的每一个标量部分初始化为零。这样的数据对象一直存在到程序结束。
- 程序在每一个块的入口分配*动态存储空间*。如果没有指定一个数据对象的初始值，那么它的初始内容是不确定的。这样的数据对象存在到块的执行终止。
- 只有当调用函数 calloc、malloc 或者 realloc 时，程序才分配*可控的*（allocated）*存储空间*。只有调用 calloc 时，它才会把这样的一个数据对象初始化为一个零字符串的数组。这样的一个数据对象一直存在到用它的地址作为参数调用 free 函数，或者直到程序结束。

那些对可分配的空间进行操作的函数就是 <stdlib.h> 中声明的存储分配函数。

堆

静态存储空间存在于程序的整个执行过程中。动态存储空间遵循后进先出的原则，它可以用一个栈实现。动态存储空间经常与函数调用和返回信息一起共享调用栈。（参考 8.1 节开始的讨论。）可控的存储空间不遵守这样规范的原则。程序可以按照任意顺序对这样的数据对象混合进行分配和释放。因此，C 标准库必须维持一个独立的称作堆的空间池来满足控制存储空间的要求。

在一些实现中，调用栈和堆要竞争有限的存储空间。使用 malloc 分配足够多的空间就可能限制程序以后调用函数的深度，或者可能会很容易用完堆中的空间。在任何情况下，只分配需要的空间并且在用过之后尽快释放它是很好的习惯。

堆的开销

注意，可控的存储空间必然会涉及开销问题。每一个分配的数据对象都要有足够多的信息让 free 来确定将要释放的区域的大小。分配 1000 个单字

符的数据可能很容易消耗 4 到 8 倍的堆空间。堆也容易受到碎片的影响。在堆上按照任意顺序分配和释放数据对象会不可避免地在某些分配的数据对象之间留出一些不可使用的小空间。这样就大大减少了堆的可用大小。

不要太在意这些东西。把相关的数据构建到一个结构中并一次分配所有的空间，这样肯定会使堆的开销减到最小，而且这也是一种好的编程风格。不要只为了节省堆开销而把不相关的数据聚集在一起。类似地，为那些具有相近的生存期的数据对象同时分配空间，然后几乎同时释放它们。这样就会使堆的破碎程度最小化，而且它也是良好的编程风格。不要为了减小堆破碎程度而提前或者推迟不相关的堆操作。存储分配函数对编程的灵活性有很大的帮助，要按照它们期望的方式去使用它们。

多字节字符集

另外一组相关函数可以操作大字符集。面对日文中的汉字和其他大字符集在计算机类产品中的使用的快速增长，标准 C 添加了这组函数。这些函数用两种方法表示这些字符集。

- 多字节字符是一个或者多个编码的序列，在这里每一编码都可以用 C 字符数据类型来表示。（字符数据类型有 *char*、*signed char* 和 *unsigned char*。在一个给定的实现中，它们具有相同的大小，而且大小至少是 8 位。）任何多字节编码的一个子集是基本 C 字符集，其中每个字符都是一个长度为 1 的序列。
- 宽字节字符是 wchar_t 类型的整数，wchar_t 在 <stddef.h> 和 <stdlib.h> 中都有定义。（假设 wchar_t 可以是从 *char* 到 *unsigned long* 之间的任意整数类型。）这样的整数可以表示大字符集中每一个字符的不同编码。基本 C 字符集中的编码和它们的单字符形式的值相同。

多字节字符便于程序和外部世界之间的通信。磁质存储和通信连接的发展已经足以支持 8 位字符的序列。宽字节字符便于程序内部的文本操作，它们的固定大小简化了对单个字符和字符数组的处理。

C 标准仅仅作了能够支持这两种编码的最少的定义。mblen、mbstowcs 和 mbtowc 把多字节字符转换为宽字节字符，wcstombs 和 wctomb 执行相反的操作。我们相信一些更加详细的函数将会很快被标准化。然而，现在，我们所能使用的就这么多。

用户可能不会立即打算编写可以方便处理大字符集的程序，但这不会阻止用户编写出尽可能多地容许大字符集的程序。例如，看一下这样的字符如何以 <stdio.h> 中的打印和扫描函数以及 <time.h> 中声明的 strftime 函数所使用的格式出现。

下面对 <stdlib.h> 中定义的每个宏和声明的每个函数分别加以说明。

EXIT_FAILURE　EXIT_FAILURE——用这个宏作为 exit 的参数或者 main 函数的返回值来报告不成功的程序终止。其他的任意非零值对不同的操作系统可能有不同的含义。

EXIT_SUCCESS　EXIT_SUCCESS——使用这个宏作为 exit 的参数或者 main 函数的返回值来报告成功的程序终止，也可以使用零。其他的任何值对不同的操作系统可能有不同的含义。

MB_CUR_MAX　MB_CUR_MAX——在当前的区域设置下，只定义了一个宽字节字符的多字节序列不能比 MB_CUR_MAX 长。可以声明一个大小为 MB_LEN_MAX（<limits.h> 中定义）的字符缓冲区，然后把 MB_CUR_MAX 个字符安全地存储为这个缓冲区的初始元素。用不小于 MB_CUR_MAX 的第三个参数调用 mbtowc，足以让函数确定一个有效的多字节序列中的下一个宽字节字符，参考本书后面 wctomb 的例子。

RAND_MAX　RAND_MAX——使用这个值去除 rand 的返回值的。例如，如果需要分布在 [0.0,1.0] 区间上的 float 类型的随机数，就可以使用表达式 (float)rand()/RAND_MAX。RAND_MAX 的值至少为 32 767。

size_t　size_t ——参考 11.3 节。

wchar_t　wchar_t ——参考 11.3 节。

div_t　div_t——声明一个这种类型的数据对象来存储 div 的返回值，div 将在下面描述。

ldiv_t　ldiv_t——声明一个这种类型的数据对象来存储 ldiv 的返回值，ldiv 将在后面描述。

abort　abort——只有当程序出现严重问题的时候才调用这个函数，它有效地调用第 13 章中描述的 raise(SIGABRT)，这会让 SIGABRT 的信号处理程序去完成最后的所有操作。另一方面，不能保证输入 / 输出流已经被清空，文件已经被正确关闭或者临时文件已经被删除了。在一般情况下，不要调用 abort 而调用 exit(EXIT_FAILURE)。

abs　abs——调用 abs (x) 来代替表达式 x<0?-x:x。越来越多的标准 C 翻译器为 abs 生成比上面的表达式更短而且更快的内联代码。另外，使用它还可以避免由于粗心而对一个表达式进行两次求值所造成的副作用。注意，在一个使用 2 的补码的机器中，abs 可能会造成溢出。（参考 5.4 节。）

atexit　atexit——使用这个函数注册程序快结束时调用的另一个函数。例如，用户可能创建了很多临时文件，在程序结束前希望把它们删除。可以编写函数 void tidy (void) 来移除这些文件。一旦存储了第一个要删除的文件名，就调用 atexit (&tidy)。当 main 函数返回或者有函数调用 exit 时，库就

会按照注册的反顺序调用 atexit 注册的所有函数，而且会在调用所有注册的函数后清空流、关闭文件和删除临时文件。使用 atexit 最多可以注册 32 个函数。

atof

atof——atof（s）和 strtod（s，NULL）等价，只不过 atof 不会把 ERANGE 存储在 errno 中来报告一个范围错误。（参考第 13 章）使用 atof 也不能知道 s 指向的串中有多少字符参与了转换。所以，最好使用 strtod 来代替它。

atoi

atoi——使用（int）strtol（s，NULL，10）来代替 atoi（s）。然后可以考虑修改第二个参数以确定多少字符参与了转换。其中的原因可以参考上面对 atof 的讨论。

atol

atol——使用 strtol（s，NULL，10）来代替 atol（s），其中的原因可以参考上面对 atof 和 atoi 的讨论。

bsearch

bsearch——这个函数用来搜索任意一个数组，这些数组的元素通过两两比较进行排序，也可以提供一个比较函数来定义它们的顺序。例如，可以用下面的基本形式构建一个关键字查找函数：

```
#include <stdlib.h>
#include <string.h>

typedef enum {FLOAT, INTEGER} Code;
typedef struct {
    char *s;
    Code code;
    } Entry;
Entry symtab[] = {
    {"float", FLOAT},
    {"integer", INTEGER}}

static int cmp(const void *ck, const void *ce)
    { /* compare key to table element */
    return (strcmp ( (char *) ck, ( (Entry *) ce) -s) ) ;
    }

Entry *lookup (char *key)
    { /* lookup key in table */
    return (bsearch (key, symtab,
        sizeof symtab / sizeof symtab[0],
        sizeof symtab[0], &cmp) );
    }
```

下面是一些注意事项：

- 如果关键字和两个或者更多个元素相等，bsearch 返回的指针可能指向这些元素中的任何一个。
- 当执行字符集改变时，要注意元素排序方式的改变——用一个兼容的比较函数调用 qsort（下面会讲）来保证数组被正确排序。

- 使用 <string.h> 中声明的函数 strcmp 或者 strcoll 时要小心。它们两个都要求字符串存储在待搜索的数组中。不能用它们来搜索指向字符串的指针数组。例如，要使用 strcmp，就必须编写一个形如 (int(*)(const void *,const void *))&strcmp 的函数指针参数。

calloc　calloc——使用这个函数来分配一个数组数据对象，并且在组成这个数据对象的所有字符中都存储为 0。可以假设任何字符类型的大小都是 1，但是要使用运算符 sizeof 来确定第二个参数，不能把第二个参数的值指定为 0。

为了最大限度的可移植，不要认为任何浮点值都能变为 0 或者任何指针都可以变为空指针。它们可能是这样，但并不一定是。也不能假设两个参数的乘积就是所需要的空间大小。一个实现可以根据第二个参数指定的大小为分配的数据对象选择一种存储对齐方式。因此，应该按以下形式分配空间：

- 一个有 N 个元素的整型数组用 calloc (N,sizeof (int)) 分配;
- 一个 struct x 类型的数据对象用 calloc (1, sizeof (struct x)) 分配。

div　div——调用 div 可能是由于下面的某个原因。

- 不管在具体的实现中操作符 / 和 % 的行为怎样，div 总是计算向零趋近的商和相应的余数。当其中一个操作数为负时这很重要。表达式 (-3)/2 可能产生 -2 或者 -1，而 div(-3,2).quot 总是产生 -1。类似地，(-3)%2 会得到 1 或者 -1，而 div(-3,2).rem 总是得到 -1。
- div 同时计算商和余数，当需要这两个结果时这就很方便。如果函数展开为只包括单一除法的内联代码，它可能会更有效。

注意，最后返回的结构类型 div_t 的成员可以按照任何顺序排列。不要对这种结构的表示方式做任何的假设。

exit　exit——调用 exit 可以在一个程序内部的任何地方终止程序的执行。在 main 函数中既可以调用 exit，也可以使用 return 语句。exit 的参数（或者 main 的返回值）应该为零或者上面描述的 EXIT_SUCCESS 来报告成功的终止；否则，它应该是上面描述过的 EXIT_FAILURE。

free　free——使用这个函数来释放程序前面调用 calloc、malloc 或者 realloc 分配的存储空间。可以安全地用一个空指针调用 free。（这种情况下这个函数什么也不做。）否则，free 的参数必须是以上 3 个函数中的一个返回的值 p。不要调用 free ((char *) p+N) 来释放前 N 个分配的字符之后的空间——调用 realloc (p,N)。一旦调用了 free (p)，就不要在任何表达式中访问 p 中的当前值——一些计算机体系结构可能会把这种访问作为致命的错误来对待。

用户没有义务释放分配的空间，但尽早地释放所有分配的空间是一个很好的原则。释放的空间可以被重新分配，来更好地利用有限的资源。而且，某些实现会在程序终止的时候报告分配的空间，这可以帮助用户找到由于粗心而没有释放的空间。

getenv

getenv——使用这个函数来获得指向一个环境变量相关的数值串的指针（参考 6.1 节）。如果用户命名了一个没有定义的环境变量，就会得到一个空指针作为函数的值。用户不能改变这个串值。然而，后面对 getenv 的调用会改变这个串。要分配一个私有的副本，可以编写类似下面的代码：

```
#include <stdlib.h>

char *copyenv (const char *name)
    { /* get and copy environment variable */
    char *s1 = getenv(name) ;
    char *s2 = s1 ? malloc(strlen(s1) + 1) : NULL;

    return (s2 ? strcpy (s2, s1) : NULL);
    }
```

labs

labs——参考上面对 abs 的讨论。

ldiv

ldiv——参考上面对 div 的讨论。

malloc

malloc——参考上面对 calloc 的讨论。使用 malloc 来为一个人工初始化的数据对象分配空间。如果数据对象只包括整数，并且用户希望它们都设置为零，就调用 calloc 而不用 malloc。calloc 的第二个参数的注意事项也适用于 malloc 的参数。

mblen

mblen——使用这个函数来确定只定义了一个宽字节字符的多字节序列的长度，该长度不能比 <stdlib.h> 中定义的 MB_CUR_MAX 大。多字节序列可以包含锁定转移，它能改变对后面任意数目的字符的解释。因此，mblen 在一个私有的静态数据对象中存储当前正在扫描的多字节串的转移状态。如果 mblen（NULL,0）的调用为非零，那么就可以通过重复地调用 mblen 来安全地扫描多字节串，每次只能扫描一个多字节串。例如，下面是一个检查多字节串是否有一个有效的编码的函数：

```
#include <stdlib.h>

int mbcheck(const char *s)
    { /* return zero if s is valid */
    int n;

    for (mblen(NULL, 0); ; s += n)
        if ( (n = mblen (s, MB_CUR_MAX) ) <= 0)
            return (n);
   }
```

mbstowcs mbstowcs——使用这个函数把整个多字节字符串转换为一个宽字节字符串。不需要担心是否发生了锁定转移，因为这个函数是对整个多字节串进行处理。也不用担心生成的宽字节字符串太长，因为第三个参数 n 限制了存储的元素的数目。如果函数返回了一个大于或者等于 n 的值，则这个转换是不完全的。如果函数返回一个负值，那么多字节串的编码是无效的。

mbtowc mbtowc——这个函数的使用和上面描述过的 mblen 的使用基本相同。这两个函数存在以下两点不同。

- 如果 mbtowc 的第一个参数不是一个空指针，那么函数返回它转换的宽字节字符。因此，它一次可以转换一个宽字节字符，而 mbstowcs 一次可以转换整个串。
- 函数 mblen 和 mbtowc 保持了单独的静态数据对象以存储转移状态。因此，即使当多字节串具有锁定转移的时候用户也可以使用这两个函数同时扫描不同的串。

qsort qsort——使用这个函数可以对任意一个数组进行排序，该数组的元素通过两两比较来排序。用户可以提供一个比较函数来定义它的顺序。这个比较函数的说明和上面所说的 bsearch 的比较函数的说明相似。不同的是，bsearch 的比较函数对关键字和一个数组元素进行比较，而 sort 比较函数对两个数组元素进行比较。

下面是几点注意事项。

- 不管函数的名字是什么，都不要认为它使用的是“快速排序”算法，因为可能不是。如果两个或者更多的元素相等，qsort 可能会按照任何相关顺序保留这些元素。因此，qsort 不是稳定的排序。
- 当执行字符集改变时要知道元素排序方式的改变。
- 使用 <string.h> 中声明的函数 strcmp 和 strcoll 时要小心。这两个函数都要求进行排序的串存储在数组中。不能使用它们对一个指向字符串的指针数组进行排序。例如，要使用 strcmp，必须使用类似于 (int (*)(const void *,const void *))&strcmp 的方式编写一个函数指针参数。

rand rand——调用 rand 来获得一个伪随机数序列中的下一个数。对于一个确定的参数值， srand（下面会描述）的每一次调用都会得到相同的随机数序列。这经常是人们想要的结果，特别是当调试程序的时候。如果想得到不可预见的行为，可以通过调用 <time.h> 中声明的 clock 或者 time 来获得 srand 的一个参数。rand 的行为在不同的实现中可能会变化很大。如果希望一直都能得到相同的伪随机数序列，可以复制 13.2 节的例子。

realloc

realloc——这个函数通常用来使前面分配的数据对象变得更大或者更小。如果让它变得更大，则存储在增加的部分的值没有定义。如果让它变小，则存储在保留的部分的值不变。然而，不管哪种情况，函数都有可能改变数据对象存储的地址。和 free（上面有描述）函数一样，一旦 realloc 返回，用户就不能在任何表达式中访问它的参数值。可以用 malloc (size) 来代替 realloc (NULL, size)。上面描述的 calloc 的第二个参数的注意事项也适用于 realloc 的第二个参数。

srand

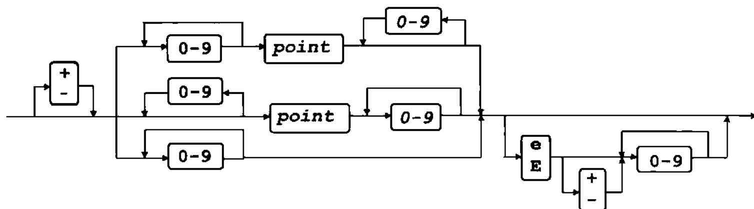
srand——参考上面对 rand 的讨论。rand 在程序启动时就调用了 srand (1)。

strtod

strtod——该函数为扫描函数（在 <stdio.h> 中声明）调用，用来把一个字符序列转换为一个 *double* 类型的编码值。可以直接调用 strtod 来避免扫描函数的开销。这样也可以更精确地确定字符串参数的哪部分参与了转换。

注意 strtod 的行为会随着区域设置而改变。这个函数调用 isspace 来跳过开头的空白。Plauger 和 Brodie 提供的图 13-1 给出了它遵循的文本模式。这里，*point* 和当前的区域设置中定义的小数点匹配。例如，这个图告诉我们，下面都是表示值 12 的有效方式：12、+12. 和 .12e2。一个实现也可以识别 "C" 之外的区域设置中的其他方式。

图 13-1
strtod 模式

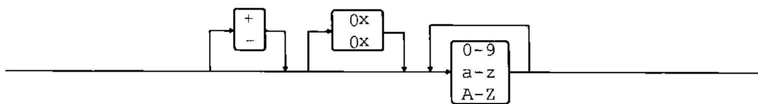


strtol

strtol——该函数为扫描函数（在 <stdio.h> 中声明）调用，用来把一个字符序列转换为一个 *long* 类型编码值。可以直接调用 strtol 来避免扫描函数的开销。这样也允许指定那些不常用的基数，并且可以更精确地确定字符串参数的哪一部分参与了转换。

注意 strtol 的行为会随着区域设置而改变。这个函数调用 isspace 来跳过开头的空白。Plauger 和 Brodie 提供的图 13-2 给出了它遵循的文本模式。例如，这个图告诉我们下面都是表示 12（假设 strtol 的第三个参数指定基数为 0）的有效形式：12、+014 和 0xC。一个实现也可以识别 "C" 之外的区域设置中的其他方式。

图 13-2
strtol 模式



strtoul strtoul——当需要 *unsigned long* 类型的结果时，可以使用这个函数而不用上面描述的 strtol。如果转换后的数值比 <limits.h> 中定义的 ULONG_MAX 大，函数 strtoul 就会报告范围错误。(取这个值的相反数不会造成溢出。) 另一方面，如果转换的结果比 LONG_MIN 小或者比 LONG_MAX 大，strtol 会报告范围错误。图 13-2 也描述了 strtoul 遵循有效的文本模式。

system system——我们不会强迫实现让 system 做任何有用的事情。如果调用 system (NULL) 返回一个非零值，用户就会知道这个函数调用了某个类别的命令处理程序。但是 C 标准对这样的创造性功能没有强加任何要求。system 的唯一可移植的用处就是提供了一种对命令处理程序的模糊访问。例如，一个编辑器可能会接受以一个感叹号开头的一行语句，它把这行中剩余的部分作为一个字符串参数传递给 system。本地的命令处理程序怎样解释这一行就不是我们关心的内容了。

wcstombs wcstombs——使用这个函数把整个宽字节字符串转换为一个多字节串。不需要担心是否发生了锁定转移，因为这个函数是对整个宽字节字符串进行处理。也不用担心生成的多字节串太长，因为第三个参数 n 限制了存储的元素的数目。如果函数返回一个比 n 大或者和 n 相等的值，那么这个转换就是不完整的。如果函数返回一个负值，那么宽字节字符串是无效的。

wctomb wctomb——使用这个函数把一个宽字节字符串转换为一个多字节串，一次转换一个宽字节字符。例如，这里有一个检查宽字节字符串的编码是否有效的函数。

```
#include <limits.h>
#include <stdlib.h>

int wccheck (wchar_t *wcs)
    { /* return zero if wcs is valid */
    char buf[MB_LEN_MAX];
    int n;

    for (wctomb(NULL, 0 ) ; ; ++wcs)
        if ( (n = wctomb (buf , *wcs) ) <= 0 )
              return (- 1);
        else if (buf [n - 1]='\0' )
             return ( 0 ) ;
}
```

注意 wctomb 返回的计数值中包括了终止的空字符，而 mbtowc 却没有。

13.4 <stdlib.h> 的实现

头文件 <stdlib.h> 就像 13.1 节指出的那样，<stdlib.h> 中声明的函数被分为 6 个不太相关的组。下面也是按照说明的顺序列出了这 6 个分组。但是我们首先来看一下头文件本身，即使它包含了很多神秘的东西。有些部分会在这里解释，本章的后面会解释其他部分。

头文件 <yvals.h> 图 13-3 显示了文件 stdlib.h。和往常一样，它包含了内部头文件 <yvals.h> 中的一些定义。其中提供了 3 个重复的定义——宏 NULL、类型 size_t 和 wchar_t（参考第 11 章）。还有一个是 <stdlib.h> 特有的——宏 _EXFAIL，这个宏确定宏 EXIT_FAILURE 的值。

宏 _EXFAIL C 标准允许每个系统为 exit（或者 main 函数的返回值）指定两个首选的参数值。宏 EXIT_FAILURE 报告失败的终止，宏 EXIT_SUCCESS 报告成功的终止。由于一些历史原因，值 0 也报告成功的终止。因此，这里只修改报告失败的终止的编码。宏 _EXFAIL 通常情况下值为 1。

数据对象 _Mbcurmax 当区域设置类别 LC_CTYPE 改变时，宏 MB_CUR_MAX 的值也会改变。它产生的值是存储在文件 xstate.c 中定义的数据对象 _Mbcurmax 中的值（参考图 6-11）

类型 _Cmpfun 我引入 _Cmpfun 类型只是为了使函数 bsearch 和 qsort 的参数声明更加简单。如果希望代码可以移植到其他实现下，那么就不要使用这个声明。（其他的私有名字会在后面解释。）

函数 abs 图 13-4 显示了文件 abs.c。绝对值函数 abs 是最简单的整型数学函数。然而，却不能提供一个屏蔽宏，因为我们必须两次访问参数的值。某些计算机体系结构下计算绝对值有特殊的指令，这就使得我们首先考虑把 abs 作为生成内联代码的内置函数处理。

函数 div 图 13-5 显示了文件 div.c，它提供了 div 函数的一个可移植的实现。如果用户知道负商向零截断，那么就可以取消测试。大多数计算机体系结构都有同时产生商和余数的除法指令，因此那些生成正确负商的函数也是内置函数的候选。一个实现可以自由地对结构体类型 div_t 的成员的顺序进行设置来使它们和硬件产生的结果相匹配。

labs ldiv 图 13-6 显示了文件 labs.c，图 13-7 显示了文件 ldiv.c。这两个文件都只是定义了函数 abs 和 div 的 *long* 类型版本。

函数 qsort 图 13-8 显示了文件 qsort.c，它定义了函数 qsort，该函数是对首元素地址为 base 的数组进行排序。函数使用的思想不太简单且有争议。它以 C.A.R.Hoare 首次提出来的快速排序算法为基础。该算法要求先选取一个枢纽元素以将数组中的元素分为两个部分，然后分别对每一部分进行排序。然后，就可以递归地使用这种技术对每个部分分别进行排序。这种算法的排序速度可能会很快，也可能会很慢。

图 13-3 stdlib.h

```
/* stdlib.h standard header */
#ifndef _STDLIB
#define _STDLIB
#ifndef _YVALS
#include <yvals.h>
#endif
        /* macros */
#define NULL          _NULL
#define EXIT_FAILURE  _EXFAIL
#define EXIT_SUCCESS  0
#define MB_CUR_MAX _Mbcurmax
#define RAND_MAX    32767
        /* type definitions */
#ifndef _SIZET
#define _SIZET
typedef _Sizet size_t;
#endif
#ifndef _WCHART
#define _WCHART
typedef _Wchart wchar_t ;
#endif
typedef struct {
    int quot;
    int rem;
    } div_t;
typedef struct {
    long quot;
    long rem;
    } ldiv_t;
typedef int _Cmpfun(const void *, const void *) ;
typedef struct {
    unsigned char _State;
    unsigned short _Wchar;
    }_Mbsave;
       /* declarations */
void abort (void) ;
int abs (int) ;
int atexit (void (*) (void) ) ;
double atof (const char *) ;
int atoi(const char *) ;
long atol (const char *) ;
void *bsearch (const void *, const void *,
    size_t, size_t, _Cmpfun *);
void *calloc (size_t , size_t) ;
div_t div (int, int) ;
void exit (int) ;
void free (void *) ;
char *getenv (const char *) ;
long labs (long) ;
ldiv_t ldiv (long, long) ;
void *malloc(size_t);
int mblen(const char *, size_t) ;
size_t mbstowcs(wchar_t *, const char *, size_t);
int mbtowc (wchar_t*, const char *, size_t) ;
void qsort (void*, size_t , size_t , _Cmpfun *) ;
```

图 13-3
（续）

```
int rand (void) ;
void *realloc (void *, size_ t) ;
void srand(unsigned int ) ;
double strtod(const char *, char **);
long strtol (const char* , char ** , int) ;
unsigned long strtoul (const char*, char ** , int) ;
int system(const char *);
size_t wcstombs (char *, const wchar_t *, size_t) ;
int wctomb (char*, wchar_t) ;
int _Mbtowc (wchar_t*, const char*, size_t , _Mbsave *) ;
double _Stod(const char *, char **) ;
unsigned long _Stoul(const char *, char **, int ) ;
int _Wctomb(char *, wchar_t, char *);
extern char _Mbcurmax , _Wcxtomb;
extern _Mbsave _Mbxlen, _Mbxtowc;
extern unsigned long _Randseed;
        /* macro overrides */
#define atof (s)  _Stod(s, 0)
#define atoi (s) (int)_Stoul (s, 0, 10)
#define atol (s) (long)_Stoul (s, 0, 10)
#define mblen(s, n) _Mbtowc(0, s, n, &_Mbxlen)
#define mbtowc(pwc, s, n)  _Mbtowc(pwc, s, n, &_Mbxtowc)
#define srand(seed)(void)(_Randseed = (seed) )
#define strtod(s, endptr) _Stod(s, endptr)
#define strtoul (s, endptr,base)  _Stoul(s,endptr , base)
#define wctomb (s, wchar) _Wctomb (s, wchar, &_Wcxtomb)
#endif
```

图 13-4
abs.c

```
/* abs function */
#include <stdlib.h>

int (abs) (int i)
     {                   /* compute absolute value of int argument */
     return ( (i < 0) ? -i : i);
     }
```

图 13-5
div.c

```
/* div function */
#include <stdlib . h>

div_t (div) (int numer, int denom)
    {                      /* compute int quotient and remainder */
    div_t val;

    val.quot = numer / denom;
    val.rem = numer - denom * val.quot;
    if (val.quot < 0 && 0 < val.rem)
       {                       /* fix remainder with wrong sign * /
       val.quot += 1;
       val.rem -= denom;
       }
    return (val);
    }
```

图 13-6
labs.c

```
/* labs function */
#include <stdlib.h>

long (labs) (long i )
    {                       /* compute absolute value of long argument */
    return ((i < 0) ? -i : i);
    }
```

图 13-7
ldiv.c

```
/* ldiv function */
#include <stdlib.h>

ldiv_t (ldiv)(long numer, long denom)
    {                       /* compute long quotient and remainder */
    ldiv_t val;

    val.quot = numer / denom;
    val.rem = numer - denom * val.quot;
    if (val.quot < 0 && 0 < val.rem)
        {                       /* fix remainder with wrong sign */
        val.quot += 1;
        val.rem -= denom;
        }
    return (val);
    }
```

图 13-8
qsort.c

```
/* qsort function */
#include <stdlib.h>
#include <string.h>

        /* macros */
#define MAX_BUF 256                         /* chunk to copy on swap */

void (qsort)( void *base, size_t n, size_t size, _Cmpfun *cmp)
    {               /* sort (char base[size]) [n] using quicksort */
    while (1 < n)
        {                                       /* worth sorting */
        size_t i = 0;
        size_t j = n - 1;
        char *qi = (char *)base;
        char *qj = qi + size*j;
        char *qp = qj;

        while (i < j)
            {                               /* partition about pivot */
            while (i < j && (*cmp)(qi, qp) <= 0)
                ++i, qi += size;
            while (i < j && (*cmp)(qp, qj) <= 0)
                --j, qj -= size;
```

图 13-8
（续）

```
        if (i < j)
            {                         /* swap elements i and j */
            char buf[MAX_BUF] ;
            char *q1 = qi;
            char *q2 = qj;
            size_t m, ms;

            for (ms = size; 0 < ms;
                ms -= m, q1 += m, q2 -= m)
                {                   /* swap as many as possible */
                m = ms < sizeof (buf) ? ms : sizeof (buf);
                memcpy(buf, q1, m);
                memcpy(q1, q2, m);
                memcpy(q2, buf, m) ;
                }
            ++i, qi += size;
            }
        }
    if (qi ! =qp)
        {                          /* swap elements i and pivot */
        char buf[MAX_BUF];
        char *q1 = qi;
        char *q2 = qp;
        size_t m, ms;

        for (ms = size; 0 < ms; ms -= m, q1 += m, q2 -= m)
            {                      /* swap as many as possible */
            m = ms < sizeof (buf) ? ms : sizeof (buf);
            memcpy(buf, q1, m);
            memcpy(q1, q2, m);
            memcpy(q2, buf, m);
            }
        }
     j = n - i - 1, qi += size;
     if (j < i)
         {                       /* recurse on smaller partition */
         if (1 < j)
             qsort (qi, j, size, cmp);
         n = i;
         }
      else
          {                        /* lower partition is smaller */
          if (1 < i)
              qsort(base, i, size, cmp);
          base = qi;
          n = j;
          }
      }
}
```

13

图 13-9
bsearch.c

```
/* bsearch function */
#include <stdlib.h>

void *(bsearch)(const void *key, const void *base,
    size_t nelem, size_t size, _Cmpfun *cmp)
    {                           /* search sorted table by binary chop */
    const char *p;
    size_t n;

    for (p = (const char *)base, n = nelem; 0 < n; )
        {                       /* check midpoint of whatever is left */
        const size_t pivot = n >> 1;
        const char *const q = p + size * pivot;
        const int val = (*cmp)(key, q);

        if (val < 0 )
            n = pivot;                          /* search below pivot */
        else if (val == 0)
            return ((void *)q);                             /* found */
        else
            {                                   /* search above pivot */
            p = q + size;
            n -= pivot + 1;
            }
        }
    return (NULL) ;                                      /* no match */
    }
```

怎样选择最好的枢轴元素是一个有争议的话题。如果选择第一个元素，则已经排好序的数组就会耗费很多时间。如果选择最后一个元素，则一个逆序的数组会耗费很多时间。如果在选择枢轴元素上花费很多工夫，那么所有的数组都会耗费很长的时间。我选择了最后一个元素，这对基本有序的数组很有效。当然，每个人都有理由选择其他的方法。

qsort 调用它本身对两部分中较小的一个进行排序，并通过内部循环对较大的一部分进行排序。这就使对动态存储空间的要求降到了最小。最坏的情况是，每一次递归调用都会对一个大小为上次调用的一半的数组进行排序。对 N 个元素进行排序所需要的递归深度不会超过 log2(N)。（对一个有 1 000 000 个元素的数组进行排序时，最多调用 20 次。）

函数 bsearch

图 13-9 显示了文件 bsearch.c。函数 bsearch 对一个以 base 为首地址且排好序的数组执行二元搜索。这种方法很简单，但也容易出错。

函数 rand

图 13-10 显示了文件 rand.c。函数 rand 使用 C 标准推荐的算法产生一个伪随机数序列（参考 13.2 节）。这种算法有合理的特性，而且被广泛使用。随机数生成器的其中一个优点就是它的随机性。另外一个优点是可重复性，这是有讽刺意义的。用户经常需要确认在伪随机数基础上的计算是否能够得

图 13-10 rand.c

```
/* rand function */
#include <stdlib.h>

/* the seed */
unsigned long _Randseed = 1;

int (rand) (void)
    {                                   /* compute pseudo-random value */
    _Randseed = _Randseed * 1103515245 + 12345;
    return ((unsigned int)(_Randseed >> 16) & RAND_MAX) ;
    }
```

图 13-11 srand.c

```
/* srand function */
#include <stdlib.h>

void (srand)(unsigned int seed)
    {                                              /*alter the seed */
    _Randseed = seed;
    }
```

到想要的结果。这种算法执行时使用 *unsigned long* 整数来避免溢出。

函数 srand

图 13-11 显示了文件 srand.c。函数 srand 只是对 _Randseed 进行设置，_Randseed 是 rand 生成伪随机数序列的种子。我为 srand 提供了一个屏蔽宏，因此，头文件 <stdlib.h> 声明了 rand.c 中定义的 _Randseed。

函数 _Stoul

图 13-12 显示了文件 xstoul.c。它定义了函数 _Stoul，这个函数执行所有从文本串到编码整数的转换。这个函数的说明和 strtoul 的相同，我把它独立出来是为了使 <stdlib.h> 中定义的几个屏蔽宏能够直接调用它。（在某些环境下，名字 strtoul 可以被重新定义。）

_Stoul 的前半部分确定基数并对最高有效数字进行定位。这就涉及了去掉开头的空白、识别所有的符号和去掉类似于 0x 的所有前缀。函数然后跳过开头的所有零，这样它就可以计算出它要转换的数的有效数字的个数。不管是否发生了溢出，函数都转换所有的有效数字。对使用 *unsigned long* 的算法来说，溢出不会造成异常。

_Stoul 首先通过检测有效数字的个数来对溢出进行粗略的检查。这个版本假设一个 *unsigned long* 占据 32 位。（如果这样的整数更大的话，就改变数组 ndigs。）对每一个有效的基数，ndigs[base] 是可能发生溢出的数字个数。因此，一个比它短的序列不会发生溢出，而比它长的一定会。一个临界长度的序列需要进一步的检查。把最后一位去掉，看看是否又得到了前面累加的值（y）。如果不是，就说明发生了溢出。

图 13-12
xstoul.c

```
/* _Stoul function */
#include <stdlib.h>
#include <ctype.h>
#include <errno.h>
#include <limits.h>
#include <stddef.h>
#include <string.h>

        /* macros */
#define BASE_MAX   36                          /* largest valid base */
        /* static data */
static const char digits[] = {                       /* valid digits */
    "0123456789abcdefghijklmnopqrstuvwxyz"};
static const char ndigs[BASE_MAX+1] = {                 /* 32-bits! */
    0, 0, 33, 21, 17, 14, 13, 12, 11, 11,
    10, 10, 9, 9, 9, 9, 9, 8, 8, 8,
    8, 8, 8, 8, 7, 7, 7, 7, 7, 7,
    7, 7, 7, 7 , 7, 7 , 7,};

unsigned long _Stoul(const char *s, char **endptr, int base)
    {           /* convert string to unsigned long, with checking */
    const char *sc, *sd;
    const char *s1, *s2;
    char sign;
    ptrdiff_t n;
    unsigned long x, y;

    for (sc = s; isspace(*sc); ++sc)
        ;
    sign = *sc == '-' || *sc == '+' ? *sc++ : '+';
    if (base < 0 || base == 1 || BASE_MAX < base)
        {                                            /* silly base */
        if (endptr)
            *endptr = (char *)s;
        return (0);
        }
    else if (base)
        {                                        /* strip 0x or 0X */
        if (base == 16 && *sc == '0'
            && (sc[1] == 'x' || sc[1] == 'X'))
            sc += 2;
        }
    else if (*sc != '0')
        base = 10;
    else if (sc[1] == 'x' || sc[1] == 'X')
        base = 16, sc += 2;
    else
        base = 8;
    for (s1 = sc; *sc == '0'; ++sc)
        ;                                    /* skip leading zeros */
    x = 0;
```

图 13-12（续）

```
    for (s2 = sc; (sd = memchr (digits,
        tolower (*sc) , base)) != NULL; ++sc)
        {                                   /* accumulate digits */
        y = x;                              /* for overflow checking */
        x = x*base + (sd - digits);
        }
    if (s1 == sc)
        {                                   /* check string validity */
        if (endptr)
            *endptr = (char *)s;
        return (0);
        }
    n = sc - s2 - ndigs[base];
    if (n < 0)
        ;
    else if (0 < n || x < x - sc[-1]
        || (x - sc[-1]) / base != y)
        {                                   /* overflow */
        errno = ERANGE;
        x = ULONG_MAX;
        }
    if (sign == '-')                        /* get final value */
        x = -x;
    if (endptr)
        *endptr = (char *)sc;
    return (x);
    }
```

图 13-13 atoi.c

```
/* atoi function */
#include <stdlib.h>

int (atoi) (const char *s)
    {                                   /* convert string to int */
    return ((int)_Stoul(s, NULL, 10)) ;
    }
```

图 13-14 atol.c

```
/* atol function */
#include <stdlib.h>

long (atol) (const char *s)
    {                                   /* convert string to long */
    return ((long)_Stoul(s, NULL, 10));
    }
```

图 13-15 strtoul.c

```
/* strtoul function */
#include <stdlib.h>

unsigned long (strtoul)(const char *s, char **endptr, int base)
    {           /* convert string to unsigned long, with checking */
    return (_Stoul(s, endptr, base));
    }
```

13

图 13-16 strtol.c

```
/* strtol function */
#include <ctype.h>
#include <errno.h>
#include <limits.h>
#include <stdlib.h>

long (strtol)(const char *s, char **endptr, int base)
    {                       /* convert string to long, with checking */
    const char *sc;
    unsigned long x;

    for (sc = s; isspace (*sc) ; ++sc)
        ;
    x = _Stoul(s, endptr, base) ;                           /* not sc! */
    if (*sc == '-' && x <= LONG_MAX)
        {                              /* negative number overflowed */
        errno = ERANGE;
        return (LONG_MIN) ;
        }
    else if (*sc !='-' && LONG_MAX < x)
        {                              /* positive number overflowed */
        errno = ERANGE;
        return (LONG_MAX) ;
        }
    else
        return ((long)x);
    }
```

图 13-17 atof.c

```
/* atof function */
#include <stdlib.h>

double (atof) (const char *s)
    {                                   /* convert string to double */
    return (_Stod(s, NULL));
    }
```

图 13-18 strtod.c

```
/* strtod function */
#include <stdlib. h>

double (strtod)(const char *s, char **endptr)
    {                     /* convert string to double, with checking */
    return (_Stod(s, endptr));
    }
```

注意 <stddef.h> 中定义的很少使用的类型 ptrdiff_t。它保证了 n 可以是两个指针之间的有符号的差值。就像 11.3 节警告的那样，ptrdiff_t 不是一个完全安全的类型。有效数字大于 32 767 的参数串在 16 位指针的计算机上可能不会报告溢出。这种可能性很小，但它还是会发生的。而且，编写完全安全的测试也很枯燥沉闷。这里我选择了速度而不是绝对安全。

atoi
atol
strtoul

图 13-13 到图 13-15 显示了文件 atoi.c、atol.c 和 strtoul.c。这些文件都定义了直接调用 _Stoul 的函数。注意，atoi 和 atol 可能溢出，C 标准对这样的溢出报告或者处理没有作任何要求。

函数 strtol

图 13-16 显示了文件 strtol.c。它定义了函数 strtol，这个函数一定会正确地报告溢出。因此，它会找出开头的所有负号，从而它能够把要转换的值作为 *long* 类型来检查。注意，这个函数必须用原始的指针调用 _Stoul。如果 _Stoul 找到一个无效的串，它一定要把那个指针存储在 endptr 中。跳过开头的任何空白都可能会造成误导。

atof
strtod

浮点数转换遵循一个相似的模式。图 13-17 显示了文件 atof.c，图 13-18 显示了文件 strtod.c。这两个函数都只是调用公共函数 _Stod 来完成所有的工作。在这种情况下，atof 要进行和 strtod 的要求完全相同的检查。

函数 _Stod

图 13-19 显示了文件 xstod.c。它定义了函数 _Stod，这个函数执行了从文本串到编码浮点数的所有转换。它执行得非常谨慎，可以避免中间数溢出和精度丢失。

例如，宏 SIG_MAX 表示了很谨慎的折中。它把有效值的位数限制在 32 位。这对这个实现支持的最精确的表示来说已经很多了（IEEE 754 10 字节的 *long double* 型大约 20 位小数）。它也远远达不到和它一致的实现中可能造成溢出的最大的整数（大约 37 位）。这个函数对任何指数的累加也很谨慎。因此，任何浮点数的上溢或者下溢都被 "xmath.h" 中声明的函数 _Dtento 安全处理了。（参考图 7-37 的文件 sdtento.c。）

这个函数的前半部分检查语法并累加有效的小数位。然后，它一次转换 8 位，转换为一个 *long* 型的数组。它把这些元素转换为 *double* 类型，从最少的有效值到最多的有效值，并且在把它们加到当前的总和之前进行适当的比例变换。这个操作序列的效率相当高而且能保持精度。

mbtowc
mblen

现在让我们看看多字节函数。图 13-21 显示了文件 mbtowc.c，图 13-20 显示了文件 mblen.c。mbtowc 和 mblen 都调用了内部函数 _Mbtowc 来完成实际的工作。每一个函数都提供了 <stdlib.h> 中定义的 _Mbsave 类型的单独的存储空间，用来记录遍历多字节串的过程中的转移状态。数据对象 _Mbxlen 和 _Mbxtowc 都有具有外部连接的名字。这样就允许头文件 <stdlib.h> 为这两个函数定义屏蔽宏。原则上，mblen 比 mbtowc 简单。然而，在这个实现中，这两个函数必须做的工作几乎没有什么差别。

函数 mbstowcs

图 13-22 显示了文件 mbstowcs.c。函数 mbstowcs 重复地调用 _Mbtowc 来完成整个多字节串到一个宽字节字符串的转换。它也提供了 _Mbsave 类型的存储空间，但是不用保留两个调用之间的转移状态。

图 13-19
xstod.c

```
/* Stod function */
#include <ctype.h>
#include <float.h>
#include <limits.h>
#include <locale.h>
#include <stdlib.h>
#include "xmath.h"

#define SIG_MAX 32

double _Stod(const char *s, char **endptr)
    {                       /* convert string to double, with checking */
    const char point = localeconv()->decimal_point[0];
    const char *sc;
    char buf[SIG_MAX], sign;
    double x;
    int ndigit, nsig, nzero, olead, opoint;

    for (sc = s; isspace(*sc); ++sc)
        ;
    sign = *sc == '-' || *sc == '+' ? *sc++ : '+';
    olead = -1, opoint = -1;
    for (ndigit = 0, nsig = 0, nzero = 0; ; ++sc)
        if (*sc == point)
            if (0 <= opoint)
                break;                      /* already seen point */
            else
                opoint = ndigit;
        else if (*sc == '0')
            ++nzero, ++ndigit;
        else if (!isdigit(*sc))
            break;
        else
            {                               /* got a nonzero digit */
            if (olead < 0)
                olead = nzero;
            else                                  /* deliver zeros */
                for (; 0 < nzero && nsig < SIG_MAX; --nzero)
                    buf[nsig++] = 0;
            ++ndigit ;
            if (nsig < SIG_MAX)                   /* deliver digit */
                    buf[nsig++] = *sc - '0';
                }
    if (ndigit == 0)
        {                                            /* set endptr */
        if (endptr)
            *endptr = (char *)s;
        return (0.0);
        }
    for (; 0 < nsig && buf[nsig - 1] == 0; --nsig)
        ;                               /* skip trailing digits */
```

图 13-19（续）

```
    {                                           /* compute significand */
    const char *pc = buf;
    int n;
    long lo[SIG_MAX/8+1];
    long *pl = &lo[nsig >> 3];
    static double fac[] = {0, 1e8, 1e16, 1e24, 1e32};

    for (*pl = 0, n = nsig; 0 < n; --n)
        if ((n & 07) == 0)                          /* start new sum */
            *--pl = *pc++;
        else
            *pl = *pl* 10 + *pc++;
    for (x = (double)lo[0], n = 0; ++n <= (nsig >> 3); )
        if (lo[n] != 0)
            x += fac[n] * (double)lo[n];
    }
    {                                   /* fold in any explicit exponent */
    long lexp = 0;
    short sexp;

    if (*sc == 'e' || *sc == 'E')
        {                                       /* parse exponent */
        const char *scsav = sc;
        const char esign = *++sc == '+' || *sc == '-'
            ? *sc++ : '+';

        if (!isdigit(*sc))
            sc = scsav;                         /* ill-formed exponent */
        else
            {                               /* exponent looks valid */
            for (; isdigit(*sc) ; ++sc)
                if (lexp < 100000)                  /* else overflow */
                    lexp= lexp * 10 + *sc - '0';
            if (esign == '-')
                lexp = -lexp;
            }
        }
    if (endptr)
        *endptr = (char *) sc;
    if (opoint < 0)
        lexp += ndigit - nsig;
    else
        lexp += opoint - olead - nsig;
    sexp = lexp < SHRT_MIN ? SHRT_MIN : lexp < SHRT_MAX
        ? (short)lexp : SHRT_MAX;
    x = _Dtento (x, sexp);
    return (sign == '-' ? -x : x);
    }
}
```

□

图 13-20
mblen.c

```
/* mblen function */
#include <stdlib.h>

        /* static data */
_Mbsave _Mbxlen = {0};

int (mblen) (const char *s, size_t n)
    {                   /* determine length of next multibyte code */
    return (_Mbtowc(NULL, s, n, &_Mbxlen));
    }
```

图 13-21
mbtowc.c

```
/* mbtowc function */
#include <stdlib.h>

        /* static data */
_Mbsave _Mbxtowc = {0};

int (mbtowc)(wchar_t *pwc, const char *s, size_t n)
    {                          /* determine next multibyte code */
    return (_Mbtowc(pwc, s, n, &_Mbxtowc));
    }
```

图 13-22
mbstowcs.c

```
/* mbstowcs function */
#include <stdlib.h>

size_t (mbstowcs) (wchar_t *wcs, const char *s, size_t n)
    {            /* translate multibyte string to wide char string */
    int i;
    wchar_t *pwc;
    _Mbsave state = {0};

    for (pwc = wcs; 0 < n; ++pwc, --n)
        {                       /* make another wide character */
        i = _Mbtowc(pwc, s, n, &state);
        if (i == -1)
            return (-1);
        else if (i == 0 || *pwc == 0)
            return (pwc - wcs);
        s += i;
        }
    return (pwc - wcs);
    }
```

函数 _Mbtowc

图 13-23 显示了文件 xmbtowc.c。函数 _Mbtowc 对一个多字节序列进行充分的分析，以此来确定它要表示的下一个宽字节字符。它是作为一个有限状态机，来完成此项操作的，有限状态机执行存储在 _Mbstate（文件 xstate.c 中定义）的状态表。（参考图 6-11。）

_Mbtowc 一定要特别严密，因为 _Mbstate 可能会有缺陷。它可能会随着区域设置类别 LC_CTYPE 而改变，且改变方式是 C 标准库无法控制的。

图 13-23 xmbtowc.c

```
/* _Mbtowc function */
#include <limits.h>
#include <stdlib.h>
#include "xstate.h"

int _Mbtowc(wchar_t *pwc, const char *s, size_t nin,
    _Mbsave *ps)
    {                           /* translate multibyte to widechar */
    static const _Mbsave initial = {0};

    if (s == NULL)
        {                                   /* set initial state */
        *ps = initial;
        return (_Mbstate._Tab[0] [0] & ST_STATE);
        }
     {                              /* run finite state machine */
    char state = ps->_State;
    int limit = 0;
    unsigned char *su = (unsigned char *)s;
    unsigned short wc = ps->_Wchar;

    if (MB_CUR_MAX < nin)
        nin = MB_CUR_MAX;
    for (; ; )
        {                       /* perform a state transformation */
        unsigned short code;
        const unsigned short *stab;

        if (_NSTATE <= state
            || (stab = _Mbstate._Tab[state]) == NULL
            || nin == 0
            || (_NSTATE*UCHAR_MAX) <= ++limit
            || (code = stab[*su]) == 0)
            break;
        state = (code & ST_STATE) >> ST_STOFF;
        if (code & ST_FOLD)
            wc = wc & ~UCHAR_MAX | code & ST_CH;
        if (code & ST_ROTATE)
            wc = wc >> CHAR_BIT & UCHAR_MAX | wc << CHAR_BIT;
        if (code & ST_INPUT && *su != '\0')
            ++su, --nin, limit = 0;
        if (code & ST_OUTPUT)
            {                           /* produce an output wchar */
            if (pwc)
                *pwc = wc;
            ps->_State = state;
            ps->_Wchar = wc;
            return ((const char *)su - s);
            }
        }
    ps->_State = _NSTATE;                         /* error return */
    return (-1);
     }
    }
```

13

要注意函数选择错误返回的原因有以下几种。

- 如果转移到一个未定义的状态。
- 如果一个给定的状态没有相应的状态表。
- 如果一个多字节串在一个多字节字符的中间部分转换结束。
- 如果函数生成一个宽字节字符时要做太多的状态转换以致它必须通过循环实现。
- 如果状态表入口专门报告了一个错误。

_Mbtowc 的剩余部分相比较而言就很简单了。函数保留宽字节字符累加器(ps->_Wchar)作为状态记录的一部分。这样就使得在一个给定的转移状态中用一个常用的组成部分生成一个宽字节字符序列变得更简单。_Mbtowc 在传送每一个宽字节字符之后返回。

函数 wctomb 图 13-24 显示了文件 wctomb.c。函数 wctomb 只是调用内部函数 _Wctomb 来提供独立的状态记录。在这种情况下，转移状态可以存储在一个 *char* 类型的数据对象中。数据对象 _Wcxtomb 有一个具有外部连接的名字，所以头文件 <stdlib.h> 可以为 wctomb 定义一个屏蔽宏。

函数 wcstombs 图 13-25 显示了文件 wcstombs.c。函数 wcstombs 反复地调用 _Wctomb 来把一个宽字节字符串转换为一个多字节串。它也提供了自己的状态记录，但不需要记录两个调用之间的转移状态。

使这个函数变得复杂的是它写入的 *char* 数组的有限长度。如果至少可以剩余 MB_CUR_MAX 个元素，那么 _Wctomb 就能直接传送字符。否则，wcstombs 必须把生成的字符存储在一个长度为 MB_CUR_MAX 的数组中，并且传送尽可能多的字符。

函数 _Wctomb 图 13-26 显示了文件 xwctomb.c。函数 _Wctomb 用来把一个宽字节字符转换为组成它的多字节表示的一个或者多个字符。它是作为一个有限状态机来完成这项操作的，该有限状态机执行存储在 _Wcstate（在文件 xstate.c 中定义）中的状态表。（参考图 6-11。）

_Wctomb 一定要非常严密，因为 _Wcstate 也可能会有缺陷。它可能会随着区域设置类别 LC_CTYPE 而改变，且改变方式是 C 标准库无法控制的。注意函数选择错误返回的原因有以下几种。

- 如果转移到一个未定义的状态。
- 如果一个给定的状态没有相应的状态表。
- 如果生成的多字节串可能会变得比 MB_CUR_MAX 个字符还长。
- 如果函数生成一个宽字节字符时要做太多的状态转换以致它必须通过循环实现。
- 如果状态表入口专门报告了一个错误。

相比之下，_Wctomb 的剩余部分也同样简单。它在处理完每一个宽字节字符输入之后返回。

图 13-24
wctomb.c

```
/* wctomb function */
#include <stdlib.h>

        /* static data */
char _Wcxtomb = {0};

int (wctomb) (char *s, wchar_t wchar)
    {               /* translate wide character to multibyte string */
    return (_Wctomb(s , wchar, &_Wcxtomb));
    }
```

图 13-25
wcstombs.c

```
/* wcstombs function */
#include <limits.h>
#include <string.h>
#include <stdlib.h>

size_t (wcstombs)(char *s, const wchar_t *wcs, size_t n)
    {             /* translate wide char string to multibyte string */
    char *sc;
    char state = {0};
    size_t i;

    for (sc = s ; 0 < n ; n -= i , ++wcs)
        {                       /* translate another wide character */
        if (MB_CUR_MAX <= n)
            {                                     /* copy directly */
            if (( i = _Wctomb(sc, *wcs, &state)) <= 0)
                return (-1);
            }
        else
            {                            /* copy into local buffer */
            char buf[MB_LEN_MAX];

            if ((i = _Wctomb(buf, *wcs, &state)) <= 0)
                return (-1);
            else if ( i <= n)
                memcpy(sc, buf, i );
            else
                {                                 /* won't all fit */
                memcpy(sc, buf, n);
                return (sc - s + n);
                }
            }
        sc += i;
        if (sc[-1] == '\0')
            return (sc - s - 1);
        }
    return (sc - s);
    }
```

13

图 13-26
xwctomb.c

```
/* _Wctomb function */
#include <limits.h>
#include <stdlib.h>
#include "xstate.h"

int _Wctomb (char *s, wchar_t wcin , char *ps)
    {                           /* translate widechar to multibyte */
    static const char initial = {0};

    if (s == NULL)
        {                                      /* set initial state */
        *ps = initial;
        return (_Mbstate._Tab[0] [0] & ST_STATE);
        }
     {                                 /* run finite state machine */
    char state = *ps;
    int leave = 0;
    int limit = 0;
    int nout = 0;
    unsigned short wc = wcin;

    for (; ; )
        {                       /* perform a state transformation */
        unsigned short code;
        const unsigned short *stab;

        if (_NSTATE <= state
            || (stab = _Wcstate._Tab[state]) == NULL
            || MB_CUR_MAX <= nout
            || (_NSTATE*UCHAR_MAX) <= ++limit
            || (code = stab[wc & UCHAR_MAX]) == 0)
            break;
        state = (code & ST_STATE) >> ST_STOFF;
        if (code & ST_FOLD)
            wc = wc & ~UCHAR_MAX | code & ST_CH;
        if (code & ST_ROTATE)
            wc = wc >> CHAR_BIT & UCHAR_MAX | wc << CHAR_BIT;
        if (code & ST_OUTPUT)
            {                              /* produce an output char */
            if ((s[nout++] = code & ST_CH ? code : wc) == '\0')
                leave = 1;
            limit = 0;
            }
        if (code & ST_INPUT || leave)
            {                                      /* consume input */
            *ps = state;
            return (nout);
            }
        }
    *ps = _NSTATE;
    return (-1);
     }
    }
```
□

图 13-27 xalloc.h

```
/* xalloc.h internal header */
#include <stddef.h>
#include <stdlib.h>
#ifndef _YVALS
#include <yvals.h>
#endif
        /* macros */
#define CELL_OFF   (sizeof (size_t) + _MEMBND & ~_MEMBND)
#define SIZE_BLOCK 512                          /* minimum block size */
#define SIZE_CELL  \
    ((sizeof (_Cell) + _MEMBND & ~_MEMBND) - CELL_OFF)
        /* type definitions */
typedef struct _Cell {
    size_t _Size;
    struct _Cell *_Next;
    } _Cell;
typedef struct {
    _Cell **_Plast;
    _Cell *_Head;
    } _Altab;
        /* declarations */
void *_Getmem(size_t);
extern _Altab _Aldata;
```

存储分配

在程序的执行过程中，几个函数合作来完成空间的分配和释放。这些函数有很多实现方法。我选择将可用空间池（"堆"）作为一个单链表来维护，链表元素仍是按照它们在存储空间中的地址排列的，一个静态的指针指向表的表头——最低地址的元素。

头文件 "xalloc.h"

图 13-27 显示了文件 xalloc.h，它是一个被所有的空间分配函数包含的内部头文件。该文件定义了一些宏和类型。例如，一个表的元素类型为 _Cell。至少该表要以这样的一个数据对象开头。成员 _Size 以字节为单位给出了整个元素的大小，它一般比一个 _Cell 数据对象要大得多。成员 _Next 指向可用存储表的下一个元素。

宏 CELL_OFF

一个已经分配的元素仍以成员 _Size 开头。后面程序想要释放这个已经分配的元素时，可能会用到该信息。然而，程序看不到这个有关大小的信息。分配函数返回一个指向成员 _Size 之外的可用空间的指针。宏 CELL_OFF 给出从已经分配的元素的起始位置开始的可用区域偏移量（以字节为单位）。

存储空间边界

很多计算机体系结构都很关心存储空间边界。一些（体系结构）要求某些类型的数据对象在字节的某些倍数的地址开始存储。典型的倍数是 2、4 或者 8 倍。其他的计算机体系结构没有要求这样的对齐，但是当对适当对齐了的数据对象进行操作时执行得更快。<stdarg.h> 中定义的宏一定要更正由于参数数据对象的对齐而留下的空隙。（参考第 10 章。）

宏 _MEMBND　存储分配函数也受存储空间边界的影响。它们认为存在最坏情况的存储空间边界，因此任何在这个边界对齐的数据对象能适当对齐。内部头文件 <yvals.h> 定义了宏 _MEMBND 来指定这种最坏情况的存储空间边界。对于一个 2^N 的边界来说，这个宏的值为 2^{N-1}。例如，在 Intel 80X86 计算机上，这个值可以是零（没有任何限制）。也许它应该至少为 1（2 字节边界）。对这样的一个具有 32 位内存的计算机，用户可能会希望让它为 3（4 字节边界）。

CELL_OFF
SIZE_CELL　存储空间分配函数的很多难以理解的逻辑都源于对最坏情况的存储空间边界进行参数化的尝试。宏 CELL_OFF 假设一个表元素从最坏情况的存储边界开始。它确定了作为下一个这样的边界的可用空间的起始位置，这个边界在为成员 _Size 留出的空间之外。同样，宏 SIZE_CELL 为一个表元素生成一个 _Size 的最小允许值。这个表元素要足够大来存储一个 _Cell 数据对象。它也必须以一个最坏情况的存储空间边界结束。

函数 malloc　和函数 malloc 一样，"xalloc.h" 的剩余部分也可以得到最好的解释。图 13-28 显示了文件 malloc.c。函数 malloc 为 size 字节的数据对象分配空间。为了实现这个功能，它从具有可用存储空间的表中寻找一个至少有 size 字节可用区域的元素。如果找到了一个，它就把超过的足以生成另一个表元素的部分分离出来。它返回一个指向可用空间的指针。

数据对象 _Aldata　malloc.c 中定义的内部函数 findmem 对可用空间表进行扫描。它在 _Altab 类型（"xstdio.h" 中定义）的数据对象 _Aldata 中保留两个静态指针。

- _Head 指向表的表头。如果表为空，它就包含一个空指针。
- _Plast 是指向下一个要考虑的表元素的指针的地址。它可以指向 _Aldata._Head 或者一个可用的表元素的 _Next，或者它也可以是一个空指针。

在任何可能的时候，findmem 都在它上一次停止调用的地方开始扫描。这种策略通过分配整个表的使用来减少了表头部分的碎片。malloc 本身和函数 free 相互协作来维护这两个指针。

如果 findmem 不能在可用表中找到一个适当的元素，那么它就尝试获得更多的存储空间。（最初堆为空，所以开始的时候要做这一步。）它调用 "xalloc.h" 中声明的函数 _Getmem 来完成这样的工作。那个基本函数一定要返回一个指向至少为要求的大小的存储空间的指针，该存储空间在最坏情况的存储边界对齐。如果不能，就返回一个空指针。

宏 SIZE_BLOCK　"xalloc.h" 中定义的宏 SIZE_BLOCK 确定了最小的首选的表元素大小。我把它设为 512，但可以修改。findmem 首先请求使用要求的大小和 SIZE_BLOCK 中较大的一个。如果失败，它就反复地将请求的大小减半，直到

请求被满足或者对要求的大小的请求不能实现。这种策略更喜欢较大的元素大小，但是最后采用它能够得到的大小。如果请求被满足了，findmem 就让新的存储空间看起来像一个之前分配的元素。它调用 free 把空间添加到可用表中。下一次执行扫描循环时应该会发现这个空间并且使用它。

函数 _Getmem

函数 _Getmem 很大程度上依赖执行环境。在每一个操作系统下都必须大范围地修改这个原语。为了完整，这里我给出了 UNIX 下运行的 _Getmem 版本。实现头文件 <stdio.h> 的几个原语时，我也是这样做的。（参考 12.4 节。）

图 13-29 显示了文件 xgetmem.c。作为早期的 UNIX 原语，它假设存在一个名字可以改变为保留形式的 C 可调用系统服务。_Sbrk 执行分配存储空间块的 UNIX 系统服务 sbrk。注意，_Sbrk 期望得到一个 *int* 类型的参数。因此，_Getmem 必须保证一个非常大的请求可以被正确解释。

函数 calloc

图 13-30 显示了文件 calloc.c。它调用 malloc 来分配存储空间，然后把它自己的字符设为零。一个更严密的版本可能会检查那两个参数的乘积大小是否适当。

函数 free

图 13-31 显示了文件 free.c。它释放前面由 malloc 或者 realloc 分配的空间。有两个常见的编程错误会对 free 造成麻烦：

- 无效的存储会改变成员 _Size 的值；
- 一个程序用一个无效的指针调用 free。这种情况说明数据对象从来没有被分配或者它已经被释放了。

也许任何数量的检查都不能阻止有缺陷的程序破坏 free。这个版本只作了一次或者两次粗略的检查。如果 _Size 成员不是最坏情况存储边界的倍数，说明它已经被修改了或者从来没有被分配。如果要释放的元素和可用空间表中的某个存在的元素重叠了，那么它被释放了两次。这两种错误都会导致 free 返回而不去释放指定的存储空间。一个更好的版本可能会报告一个信号或者生成一个诊断。至少，它可能会在 <errno.h> 声明的 errno 中存储一个非零值。

free 的大部分工作是在可用空间表中寻找适当的地方来插入释放的元素。如果释放的元素和一个或者两个存在的表元素相邻接，这些邻接的元素就会组合在一起。这样就可以使表的碎片减少。

注意，free 修改了扫描指针 _Aldata._Plast。这是很必要的，因为存储的指针可能会指向一个已经和另一个元素合并的元素。我选择从释放的元素处开始扫描。这里是一个很容易确定的地址。这种方法也可以更统一地使用表中的存储空间，并且它尽可能地推迟重新利用释放的存储空间的时间（对有漏洞的程序的关照，值得推敲）。另一方面，任何通过调用 _Getmem 使堆增长的时候，都会降低它的效率。这是一个值得探索的领域。

图 13-28
malloc.c

```
/* malloc function */
#include "xalloc.h"
#include "yfuns.h"

        /* static data */
_Altab _Aldata = {0};                          /* heap initially empty */

static _Cell **findmem(size_t size)
    {                                                   /* find storage */
    _Cell *q, **qb;

    for (; ; )
        {                                    /* check freed space first */
        if ((qb = _Aldata._Plast) == NULL)
            {                                   /* take it from the top */
            for (qb = &_Aldata._Head; *qb;
                qb = &(*qb)->_Next)
                if (size <= (*qb)->_Size)
                    return (qb);
            }
        else
            {                               /* resume where we left off */
            for (; *qb; qb = &(*qb)->_Next)
                if (size <= (*qb)->_size)
                    return (qb);
            q = *_Aldata._Plast;
            for (qb = &_Aldata._Head; *qb != q;
                qb = &(*qb)->_Next)
                if (size <= (*qb)->_Size)
                    return (qb) ;
            }
         {                                    /* try to buy more space */
        size_t bs;
        const size_t sz = size + CELL_OFF;

        for (bs = SIZE_BLOCK; ; bs >>= 1)
            {                                 /* try larger blocks first */
            if (bs < sz)
                bs = sz;
            if ((q = _Getmem(bs)) ! = NULL)
                break;
            else if (bs == sz)
                return (NULL);                          /* no storage */
            }
        /* got storage: add to heap and retry */
        q->_Size = (bs & ~_MEMBND) - CELL_OFF;
        free((char *)q + CELL_OFF);
         }
        }
    }
```

图 13-28 (续)

```
void * (malloc)(size_t size)
    {                          /* allocate a data object on the heap */
    _Cell *q, **qb;

    if (size < SIZE_CELL)                          /* round up size */
        size = SIZE_CELL;
    size = (size + _MEMBND) & ~_MEMBND;
    if ((qb = findmem(size)) == NULL)
        return (NULL);
    q = *qb;
    if (q->_Size < size + CELL_OFF + SIZE_CELL)
        *qb = q->_Next;                          /* use entire cell */
    else
        {                               /* peel off a residual cell */
        *qb = (_Cell *)((char *)q
            + CELL_OFF + size);
        (*qb) ->_Next = q->_Next;
        (*qb) ->_Size = q->_Size - CELL_OFF - size;
        q->_Size = size;
        }
    _Aldata._Plast = qb ? qb : NULL;                  /* resume here */
    return ((char *)q + CELL_OFF);
    }
```

图 13-29 xgetmem.c

```
/* _Getmem function -- UNIX version */
#include "xalloc.h"

        /* UNIX system call */
void *_Sbrk(int);

void *_Getmem (size_t size)
    {                                      /* allocate raw storage */
    void *p;
    int isize = size;

    return (isize <= 0 || (p = _Sbrk(isize)) == (void *)-1
        ? NULL : p);
    }
```

图 13-30 calloc.c

```
/* calloc function */
#include <stdlib.h>
#include <string.h>

void *(calloc)(size_t nelem, size_t size)
    {           /* allocate a data object on the heap and clear it */
    const size_t n = nelem * size;
    char *p = malloc(n);

    if (p)
        memset(p, '\0', n);
    return (p);
    }
```

图 13-31
free.c

```
/* free function */
#include "xalloc.h"

void (free)(void *ptr)
    {                              /* free an allocated data object */
    _Cell *q;

    if (ptr == NULL)
        return;
    q = (_Cell *)((char *)ptr - CELL_OFF) ;
    if (q->_Size & _MEMBND)
        return;                                       /*bad pointer */
    if (_Aldata._Head == NULL
        || q < _Aldata._Head)
        {                                 /* insert at head of list */
        q->_Next = _Aldata._Head;
        _Aldata._Head = q;
        }
    else
        {                               /* scan for insertion point */
        _Cell *qp;
        char *qpp;

        for (qp = _Aldata._Head;
            qp->_Next && q < qp->_Next;)
            qp = qp->_Next;
        qpp = (char *)qp + CELL_OFF + qp->_Size;
        if ((char *)q < qpp)
            return;                               /* erroneous call */
        else if ((char *)q == qpp)
            {                                    /* merge qp and q */
            qp->_Size += CELL_OFF + q->)_Size;
            q = qp;
            }
        else
            {                                 /* splice q after qp */
            q->_Next = qp->_Next;
            qp->_Next = q;
            }
        }
    if (q->_Next &&
        (char *)q + CELL_OFF + q->_Size == (char *)q->_Next)
        {                                   /* merge q and q->_Next */
        q->_Size += CELL_OFF + q->_Next->_Size;
        q->_Next = q->_Next->_Next;
        }
    _Aldata._Plast = &q->_Next;          /* resume scan after freed */
    }
```
□

图 13-32 realloc.c

```
/* realloc function */
#include <string.h>
#include "xalloc.h"

void * (realloc)(void *ptr, size_t size)
	{                          /* reallocate a data object on the heap */
	_Cell *q;

	if (ptr == NULL)
		return (malloc(size));
	q = (_Cell *)((char *)ptr - CELL_OFF) ;
	if (q->_Size < size)
		{                                  /* try to buy a larger cell */
		char *const new_p = malloc (size);

		if (new_p == NULL)
			return (NULL) ;
		memcpy(new_p, ptr, q->_Size);
		free (ptr);
		return (new_p);
		}
	else if (q->_Size
		< size + CELL_OFF + SIZE_CELL)
		return (ptr);                          /* leave cell alone */
	else
		{                                      /* free excess space */
		const size_t new_n = (size + _MEMBND) & ~_MEMBND;
		_Cell *const new_q = (_Cell *)((char *)ptr + new_n);

		new_q->_Size = q->_Size - CELL_OFF - new_n;
		q->_Size = new_n;
		free((char *) new_q + CELL_OFF) ;
		return (ptr);
		}
	}
```

函数 realloc

图 13-32 显示了文件 realloc.c。如果可能的话，函数 realloc 会尽力分配一个更大的存储空间。如果值得整理已经存在存储空间，它也会尽力去整理。

这个版本并没有尽最大的努力去工作。如果要求了一个更大的存储空间，这个函数坚持在释放已经存在的空间之前分配一个新的空间。这样就不必担心在重新安排过程中保留存储在可用空间中的数据。但是它排除了一种可能性——更大的空间只有当已经存在的空间释放之后才可能使用。所以这里为一个优秀的实现人员保留了可供改进的地方。

存储分配函数非常重要，很多程序依靠它们来快速并且健壮地工作。它们对调试也能提供很有价值的帮助。因为它们非常独立，所以它们很容易作为一个独立的单元来修改。由于以上这些原因，这些函数的实现有很多种。这里我强调了性能和健壮性。当然，实现者也可能想专注于其他的目标。

abort
atexit
exit

最后一组函数通过各种各样的方式和环境交互。其中 3 个函数处理程序终止——abort、atexit 和 exit。图 13-33 到图 13-35 显示了文件 abort.c、atexit.c 和 exit.c。abort 只是简单地报告信号 SIGABRT。如果这个信号的处理程序返回，那么函数就以失败的状态退出。atexit 几乎同样简单，它只是把一个函数指针推入一个由数据对象 _Atcount 和 _Atfuns 定义的栈中。对 exit 的调用就是使这个栈的栈顶元素出栈并且调用相应的函数。

函数 _Exit

exit 也在它终止程序的执行之前关闭所有打开的文件。程序的终止方式依赖于系统。然而，通常可以调用某个函数来完成这些工作。和其他几个接口原语一起，我把这个问题放进了内部头文件 "yfuns.h" 中。它或者声明一个函数，或者定义一个叫做 _Exit 的宏，该宏接受退出状态并且终止程序的执行。例如，在 UNIX 系统下，_Exit 只是 exit 系统服务的一个可更改的名字。

函数 getenv

图 13-36 显示了文件 getenv.c。它一定知道怎样访问定义了所有的环境变量的环境表，它也一定知道怎样遍历那个表来扫描具有要求的名字的环境变量。我在这里给出的版本在 UNIX 下工作。它也可以在很多其他的操作系统下工作。

getenv 假设 _Envp 指向一个以空字符结束的字符串的序列的第一个字符串。这个序列以一个空字符串结束。这个序列中的每一个串的形式都为 name = value。如果参数串和等号前面的所有的字符匹配，那么函数返回一个指向等号后面的第一个字符的指针。我再一次把定义或者声明 _Envp 的工作留给了内部头文件 "yfuns.h"。

有些操作系统支持环境表，但不是这种形式的。其他系统支持的环境表作为一个 C 数据对象不是直接可寻址的。这两种情况都要求把字符串的值复制到一个 getenv 私有的静态缓冲区中。要这样做就必须修改这个实现中的几个函数。这几个函数假设它们可以直接调用 getenv。只有在调用对用户程序没有影响的情况下这种假设才成立。必须引入一个像 _Getenv 这样让用户提供自己的静态缓冲区的函数来保存这个串的值。我选择省略阻止进一步修改的那一层。

函数 system

图 13-37 显示了文件 system.c。它展示了函数 system 的 UNIX 版本可能会从一个 C 程序中调用一个命令处理程序的方法。和往常一样，这个函数假设存在几个具有合适的保留名字的 UNIX 系统服务。而且和往常一样，我在这里给出的版本可以改进。把路径名 "/bin/sh" 作为命令处理程序的名字是最理想的情况，也是最不好的习惯。通常使用几种更成熟的方案来对指定各种命令处理程序。这个函数也能返回程序关注的更有用的状态信息。

图 13-33 abort.c

```
/* abort function */
#include <stdlib.h>
#include <signal.h>

void (abort)(void)
    {                                           /* terminate abruptly */
    raise(SIGABRT);
    exit(EXIT_FAILURE);
    }
```

图 13-34 atexit.c

```
/* atexit function */
#include <stdlib.h>

        /* external declarations */
extern void (*_Atfuns[])(void);
extern size_t _Atcount;

int (atexit)(void (*func)(void))
    {                                  /* function to call at exit */
    if (_Atcount == 0)
        return (-1);                                /* list is full */
    _Atfuns[--_Atcount] = func;
    return (0);
    }
```

图 13-35 exit.c

```
/* exit function */
#include <stdio.h>
#include <stdlib.h>
#include "yfuns.h"

        /* macros */
#define NATS   32
        /* static data */
void (*_Atfuns[NATS])(void) = {0};
size_t _Atcount = {NATS};

void (exit)(int status)
    {                               /* tidy up and exit to system */
    while (_Atcount < NATS)
        (*_Atfuns[_Atcount++])() ;
     {                                           /* close all files */
    size_t i;

    for ( i = 0; i < FOPEN_MAX; ++i)
        if (_Files[i])
            fclose(_Files[i]);
     }
    _Exit (status);
    }
```

13

图 13-36 getenv.c

```
/* getenv function -- in-memory version */
#include <stdlib.h>
#include <string.h>
#include "yfuns.h"

char *(getenv)(const char *name)
    {                       /* search environment list for named entry */
    const char *s;
    size_t n = strlen(name);

    for (s = _Envp; *a; s += strlen(s) + 1)
        {                                       /* look for name match */
        if (!strncmp(s, name, n) && s[nl == '=')
            return ((char *)&s[n + 1]);
        }
    return (NULL);
    }
```

图 13-37 system.c

```
/* system function -- UNIX version */
#include <stdlib.h>

        /* UNIX system calls */
int _Execl(const char *, const char *, ...);
int _Fork(void);
int _Wait (int *);

int (system)(const char *s)
    {               /* send text to system command line processor */
    if (s)
        {                                       /* not just a test */
        int pid = _Fork();

        if (pid < 0)
            ;                                       /* fork failed */
        else if (pid == 0)
            {                           /* continue here as child */
            _Execl("/bin/sh", "sh", "-C", s, NULL);
            exit(EXIT_FAILURE);
            }
        else                            /* continue here as parent */
            while (_Wait(NULL) ! = pid)
                ;                       /* wait for child */
```

13.5 <stdlib.h> 的测试

图 13-38 显示了文件 tstdlib.c。这个程序对 <stdlib.h> 中声明的各种各样的函数进行测试，虽然有时候很粗略。例如，函数 getenv 和 system 可以返回任何值并能满足这个测试。剩余的函数必须至少做一些有用的事。

作为结果，程序显示了宏 RAND_MAX 和 MB_CUR_MAX 的值。它也确定了 "C" 区域设置是否支持具有转移状态的多字节串。对本书的实现，这个程序显示下面的内容：

为了显示最后一行并成功地退出，程序必须完成几件正确的事情。它必须提供可以触发调用 abort 的 SIGABRT 信号的处理程序。这个处理程序必须用成功的状态 EXIT_SUCCESS 调用 exit。并且 exit 必须调用函数 atexit 注册的处理程序 done。该处理程序必须能向标准输出流写入一行文本。所有这些都在测试有关处理程序终止的逻辑。

13.6 参考文献

Donald Knuth, *The Art of Computer Programming*, Vols. 1-3 (Reading, Mass.: Addison-Wesley, 1967 and later). 这里有丰富的算法资源，并带有完全的分析和入门指南。第一卷是 *Fundamental Algorithms*，第二卷是 *Semimumerical Algorithms*，第三卷是 *Sorting and Searching*。有些现在已经出第二版了。

书中的信息包括：

- 维护一个堆；
- 计算随机数；
- 搜索有序序列；
- 排序；
- 在不同的数制之间转换。

在修改本章中的代码之前，可以参考一下 Knuth 的观点。

Ronald F. Brender, *Character Set Issues for Ada 9X*, SEI-89-SR-17 (Pittsburgh,Pa.: Software Engineering Institute, Carnegie Mellon University,October 1989). 这是对程序设计语言中关于大字符集和多字节字符集的很多问题的优秀总结。虽然这些文档以 Ada 程序设计语言为中心，但它与 C 的相关性也非常大。

13

图 13-38 tstdlib.c

```
/* test stdlib functions */
#include <assert.h>
#include <limits.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

static void abrt (int sig)
    {                                                   /* handle SIGABRT */
    exit(EXIT_SUCCESS);
    }

static int cmp(const void *p1, const void *p2)
    {                       /* compare function for bsearch and qsort */
    unsigned char c1 = *(unsigned char *)p1;
    unsigned char c2 = *(unsigned char *)p2;

    return (*(unsigned char *)p1 - *(unsigned char *)p2);
    }

static void done (void)
    {                                           /* get control from atexit */
    puts("SUCCESS testing <stdlib.h>");
    }

int main ()
    {                      /* test basic workings of stdlib functions */
    char buf [10], *s1, *s2;
    div_t iqr;
    ldiv_t lqr;
    int i1 = EXIT_FAILURE;
    int i2 = EXIT_SUCCESS;
    int i3 = MB_CUR_MAX;
    wchar_t wcs[10];
    static char abc [] = "abcdefghijklmnopqrstuvwxyz";
    static int rmax = RAND_MAX;

    assert(32767 <= rmax);
    assert(1 <= MB_CUR_MAX && MB_CUR_MAX <= MB_LEN_MAX);
    assert((s1 = malloc(sizeof (abc))) != NULL);
    strcpy(s1, abc);
    assert((s2 = calloc(sizeof (abc) , 1)) != NULL
        && s2[0] == '\0');
    assert(memcmp(s2, s2 + 1, sizeof (abc) - 1) == 0);
    assert(strcmp(s1, abc) == 0);
    assert((s1 = realloc (s1, 2 * sizeof (abc) - 1)) != NULL);
    strcat(s1, abc);
    assert(strrchr (s1, 'z') == s1 + 2 * strlen (abc) - 1);
    free(s2);
    assert((s1 = realloc(s1, sizeof (abc) - 3)) != NULL);
    assert(memcmp (s1, abc, sizeof (abc) -3) == 0);
    assert(getenv("ANY") || system(NULL) || abc[0]);
    assert(abs(-4) == 4 && abs(4) == 4);
    assert(labs(-4) == 4 && labs(4) == 4);
```

图 13-38
(续)

```
assert(div(7, 2).quot == 3 && div(7, 2).rem == 1);
iqr = div(-7, 2);
assert(iqr.quot == -3 && iqr.rem == -1);
assert(ldiv(7, 2).quot == 3 && ldiv(7, 2).rem == 1);
lqr = ldiv(-7, 2);
assert(lqr.quot == -3 && lqr. rem == -1);
assert (0 <= (i1 = rand()) && i1 <= RAND_MAX);
assert (0 <= (i2 = rand()) && i2 <= RAND_MAX);
srand(1);
assert(rand() == i1 && rand() == i2);
assert(bsearch("0", abc, sizeof (abc) - 1, 1, &cmp)
    ==NULL) ;
assert(bsearch("d", abc, sizeof (abc) - 1, 1, &cmp)
    == &abc[3]);
qsort(strcpy(buf, "mishmash"), 9, 1, &cmp);
assert(memcmp(buf, "\0ahhimmss", 9) == 0);
assert(atof("3.0") == 3.0);
assert(atof("-1e-17-") == -1e-17);
assert(atoi("37") == 37 && atoi("-7192X") == -7192);
assert(atol("+29") == 29 && atol("-077") == -77);
assert(strtod("28G", &s1) == 28.0
    && s1 != NULL && *s1 == 'G');
assert(strtol("-a0", &s1, 11) == -110
    && s1 != NULL && *s1 == '\0');
assert(strtoul("54", &s1, 4) == 0
    && s1 != NULL && *s1 == '5');
assert(strtoul("0xFfg", &s1, 16) == 255
    && s1 != NULL && *s1 =='g');
assert(mbstowcs(wcs, "abc", 4) == 3 && wcs[1] == 'b');
assert(wcstombs(buf, wcs, 10) == 3
    && strcmp (buf, "abc") == 0);
mblen(NULL, 0);
wctomb(NULL, 0);
assert(mblen("abc", 4) == 1);
assert(mbtowc(&wcs[0], "abc", 4) == 1 && wcs[0] == 'a');
assert(wctomb(buf, wcs[0]) == 1 && buf[0] == 'a');
assert(mblem("", 1) == 0);
assert(mbtowc(&wcs[0], "", 1) == 0 && wcs[0] == 0);
assert(wctomb(buf, wcs[0]) == 1 && buf[0] == '\0');
printf("RAND_MAX = %ld\n", (long)RAND_MAX);
printf("MB_CUR_MAX = %u\n", MB_CUR_MAX);
printf("Multibyte strings%s have shift states\n",
    mbtowc(NULL, NULL, 0) ? "" : " don't");
atexit(&done);
signal(SIGABRT, &abrt);
abort();
puts("FAILURE testing <stdlib.h>");
return (EXIT_FAILURE);
}
```
□

13

13.7 习题

13.1 下面的区域设置函数文件定义了日文中的汉字的多字节编码 "Shift JIS"。一个在区间 [0x81,0x9F] 或者 [0xE0, 0xFC] 内的字符编码会对一个双字符序列的第 1 个字符发出信号。（任何其他的编码都是一个字符。）第二个字符一定在区间 [0x40, 0xFC] 中。

```
LOCALE SHIFT_JIS
NOTE JIS codes with 0x81-0x9F or 0xE0-0xFC followed by 0x40-0xFC
SET A 0x81
SET B 0x9f
SET C 0xe0
SET D 0xfc
SET M 0x40
SET N 0xfc
SET X 0
mb_cur_mex 2
mbtowc[0, 0:$#] $@ $F    $O $I $0
mbtowc[0, A:B ] $@ $F $R    $I $1
mbtowc[0, C:D ] $@ $F $R    $I $1
mbtowc[1, 0:$#]    X
mbtowc[1, M:N ] $@ $F    $O $I $2
mbtowc[2, 0:$#]  0 $F $R       $0
wctomb[0, 0:$#]       $R       $1
wctomb[1, 0:$#]    X
wctomb[1, 0   ]       $R $O $I $0
wctomb[1, A:B ] $@    $R $O    $2
wctomb[1, C:D ] $@    $R $O    $2
wctomb[2, 0:$#]    X
wctomb[2, M:N ]          $O $I $0
LOCALE end
```

描述一下这个区域设置文件定义的多字节字符和宽字节字符之间的映射关系。画出 wctomb 和 mbtowc 的状态转换图。

13.2 EUC（Extended UNIX Code，扩展 UNIX 编码）中的一个定义和 Shift JIS 很相似。区间 [0xA1, 0xFE] 中的一个字符编码是一个双字符序列的第一个，第二个字符一定在区间 [0x80, 0xFF] 中。修改上题中的区域设置文件来定义这个多字节编码。描述一下你是怎样实现到宽字节字符的映射的。

13.3 下面的区域设置文件定义了具有锁定转移状态的“JIS”多字节编码。3 字符序列“\33$B”转换为双字符模式。3 字符序列“\33(B”转换回单字符模式。在双字符模式下，这两个字符编码一定都在区间 [0x21,0x7E] 内。

```
LOCALE JIS
NOTE JIS codes with ESC+(+B and ESC+$+B
SET A 0x21
SET B 0x7e
SET X 0
SET Z 033
mb_cur_max 5
```

```
mbtowc[0, 0:$#] $@ SF    $O $I $0
mbtowc[0, 0   ] $@ SF    $O $I $1
mbtowc[0, Z   ]             $I $1
mbtowc[1, 0:$#]    X
mbtowc[1, '(' ]             $I $2
mbtowc[1, '$' ]             $I $3
mbtowc[2, 0:$#]    X
mbtowc[2, 'B' ] 0  $F $R    $I $0
mbtowc[3, 0:$#]    X
mbtowc[3, 'B' ]             $I $4
mbtowc[4, 0:$#]    X
mbtowc[4, Z   ]             $I $1
mbtowc[4, A:B ] $@ $F $R    $I $5
mbtowc[5, 0:$#]    X
mbtowc[5, A:B ] $@ $F    $O $I $4
wctomb[0, 0:$#]       $R       $1
wctomb[1, 0:$#]    X
wctomb[1, 0   ]       $R $O $I $0
wctomb[1, A:B ] Z        $O    $2
wctomb[2, 0:$#] '$'      $O    $3
wctomb[3, 0:$#] 'B'      $O    $4
wctomb[4, 0:$#]    X
wctomb[4, 0   ] Z        $O    $7
wctomb[4, A:B ] $@    $R $O    $5
wctomb[5, 0:$#]    X
wctomb[5, A:B ]          $O $I $6
wctomb[6, 0:$#]       SR       $4
wctomb[7, 0:$#] '('      $O    $7+$1
wctomb[8, 0:$#] 'B'      $O    $1
LOCALE end
```

描述一下这个区域设置文件定义的多字节字符和宽字节字符之间的映射关系。画出 wctomb 和 mbtowc 的状态转换图。

13.4 修改存储分配函数，使它最多可以维护 8 个元素大小固定的表。在已经存在的元素表中加入一个相同大小的已经释放的项。（不用按存储地址对这些表进行排序。）否则，如果 8 个表中还有一些表没有建立，就创建一个新表。如果请求的大小和这个大小相同，就从这些表中分配一个。引入这种额外的复杂性有什么作用？

13.5 修改存储分配函数，使它可以在每一个分配的元素中既存储元素的大小又存储一个信号。可以尝试使用下面的代码：

```
p - >_Signature = p - >_Size ^ (int)p ^ 0x01234567;
```

（这个例子假设 p 和 p－>_Size 都占据 32 位，它不是可移植的代码。）检查每一个要释放的元素的信号。引入这种额外的复杂性有什么作用？

13.6 修改存储分配函数，要求所有分配的空间在程序终止前释放。需要修改 exit 吗？对存储分配函数这样使用要遵循哪些原则？引入这种额外的复杂性有什么作用？

13.7 实现你使用的 C 翻译器下的 exit、getenv 和 system。需要编写汇编代码吗?

13.8 [**难**] 修改 strtod，使它可以把输入串 Inf 转换为特殊编码 Inf，把输入串 NaN 转换为特殊编码 NaN。C 标准允许这个扩展吗？怎样修改 <locale.h> 中的代码来开启和关闭这个转换？能设计一个可以指定任意的非数编码的符号吗?

13.9 [**很难**] 修改一个 C 编译器，使它可以生成 abs、div、labs 和 ldiv 的内联代码。

第 14 章 <string.h>

14.1 背景知识

<string.h> 中声明的函数是对标准 C 的一个重要补充，它们支持 C 语言把文本作为字符数组操作的传统。其他一些语言很好地集成了对文本串的操作，SNOBOL 就是一个典型的例子。C 吸收的语言本身的所有东西就是空字符结尾的字符串字面量，例如 "abc"。C 标准库提供了所有重要的功能。这些函数可以对以下 3 种形式的串进行操作。

- 名字以 mem 开头的函数对任意的字符序列进行操作。其中一个参数（s）指向字符串的起始位置——最小下标的元素，另一个参数（n）对元素的个数进行计数。
- 名字以 strn 开头的函数对非空字符序列进行操作。参数 s 和 n 的含义和上面的相同。这些串恰好在元素 s[n] 前结束，或者在 s[i] 为零（'\0'）的第一个 i 处结束，这两个定义哪个序列较短，就是哪一个。
- 所有其他名字以 str 开头的函数对空字符结尾的字符序列进行操作。这些函数只使用参数 s 来确定串的开头。

正如我们所期望的，每一组函数都有它特定的用途。

缺点 大多数人可能不会想到这些函数有一些设计缺陷。<string.h> 中声明的函数不是共同设计努力的结果。相反，它们是由很多程序员在很长的时间里作出的贡献积累起来的。到 C 标准化开始的时候，“修整”它们已经太晚了，太多的程序对函数的行为已经有了明确的定义。其中的一些问题如下。

- 很多搜索函数在搜索失败的时候返回一个空指针。在进一步使用返回值之前，必须先找到它并且对它进行测试。指向串的尾部的指针和一个失败编码效果一样，但是在表达式中它会更有用。
- 复制函数返回一个指向目标区域的起始位置的指针。这在一个更大的表达式中有时候是有用的，但是副本的尾部的地址可以提供更有用的信息。使用后面的返回值比使用前者可以更有效地完成多重复制。

- 某些函数的名字太隐秘了。例如，strcspn 和 strpbrk 不能明确说明它们的作用。
- 这些函数的集合不完整并且不一致。例如，添加 strnlen 和 memrchr 这两个函数很合理，但是添加 strncat 有点令人吃惊。

尽管有这些缺陷，但是我发现 <string.h> 中声明的函数既重要又有用。事实上，它们中的某些函数还可以生成内联代码。很多 C 程序使用这些函数，并且经常使用。它们值得研究和优化。

14.2 C 标准的内容

<string.h>

7.11 字符串处理 <string.h>

7.11.1 串函数约定

size_t NULL

头文件 <string.h> 声明了一种类型和几个函数，并定义了一个宏，这个宏对操作字符类型的数组和其他可以作为字符数组对待的对象有用[133]。声明的类型是 size_t 且定义的宏是 NULL（两者都在 7.1.6 中说明）。确定数组长度的方法有很多种，但是所有情况下。char* 或者 void* 参数都指向数组的第一个（最低地址）字符。如果对数组的访问超出了对象的结尾，则这种行为未定义。

7.11.2 复制函数

memcpy

7.11.2.1 函数 memcpy

概述

```
#include <string.h>
void *memcpy(void *s1, const void *s2, size_t n);
```

说明

函数 memcpy 从 s2 指向的对象中复制 n 个字符到 s1 指向的对象中。如果复制发生在两个重叠的对象中，则这种行为未定义。

返回值

函数 memcpy 返回 s1 的值。

memmove

7.11.2.2 函数 memmove

概述

```
#include <string.h>
void *memmove(void *s1, const void *s2, size_t n);
```

说明

函数 memmove 从 s2 指向的对象中复制 n 个字符到 s1 指向的对象中。复制看上去就像先把 n 个字符从 s2 指向的对象复制到一个 n 个字符的临时数组中，这个临时数组和 s1、s2 指向的对象都不重叠，然后再把这 n 个字符从临时数组中复制到 s1 指向的对象中。

返回值

函数 memmove 返回 s1 的值。

strcpy

7.11.2.3 函数 strcpy

概述

```
#include <string.h>
char *strcpy(char *s1, const char *s2);
```

说明

函数 strcpy 把 s2 指向的串（包括终止的空字符）复制到 s1 指向的数组中。如果复制发生在两个重叠的对象中，则行为未定义。

返回值

函数 strcpy 返回 s1 的值。

strncpy

7.11.2.4 函数 strncpy

概述

```
#include <string.h>
char *strncpy(char *s1, const char *s2, size_t n);
```

说明

函数 strncpy 从 s2 指向的数组中复制最多 n 个字符（不复制空字符后面的字符）到 s1 指向的数组中[134]。如果复制发生在两个重叠的对象中，则行为未定义。

如果 s2 指向的数组是一个长度比 n 短的字符串，则在 s1 指向的数组后面添加空字符，直到写入了 n 个字符。

返回值

函数 strncpy 返回 s1 的值。

7.11.3 连接函数

strcat

7.11.3.1 函数 strcat

概述

```
#include. <string.h>
char *strcat(char *s1, const char "s2);
```

说明

函数 strcat 把 s2 指向的串（包括终止的空字符）的副本添加到 s1 指向的串的末尾。s2 的第一个字符覆盖 s1 末尾的空字符。如果复制发生在两个重叠的对象中，则行为未定义。

返回值

函数 strcat 返回 s1 的值。

strncat

7.11.3.2 函数 strncat

概述

```
#include. <string.h>
char *strncat(char *s1, const char *s2. size_t n);
```

说明

函数 strncat 从 s2 指向的数组中将最多 n 个字符（空字符及其后面的字符不添加）添加到 s1 指向的串的末尾。s2 的第一个字符覆盖 s1 末尾的空字符。通常在最后的结果后面加上一个空字符[135]。如果复制发生在两个重叠的对象中，则行为未定义。

返回值

函数 strncat 返回 s1 的值。

参见：函数 strlen（7.1 1.6.3）。

7.11.4 比较函数

比较函数 memcmp、strcmp 和 strncmp 返回的非零值的符号由参与比较的对象中的第一对不相等的若干字符（都解释为 unsigned char 类型）的差值的符号决定。

memcmp

7.11.4.1 函数 memcmp

概述

```
#include <string.h>
int memcmp(const void *s1, const void *s2, size_t n);
```

说明

函数 memcmp 将 s1 指向的对象的前 n 个字符和 s2 指向的对象的前 n 个字符进行比较[136]。

14

返回值

当 s1 指向的对象大于、等于或者小于 s2 指向的对象时，函数 memcmp 分别返回一个大于、等于或者小于零的整数。

strcmp

7.11.4.2 函数 strcmp

概述

```
# include <string.h>
int strcmp(const char *s1. const char *s2);
```

说明

函数 strcmp 对 s1 指向的串和 s2 指向的串进行比较。

返回值

当 s1 指向的串大于、等于或者小于 s2 指向的串时，函数 strcmp 分别返回一个大于、等于或者小于零的整数。

strcoll

7.11.4.3 函数 strcoll

概述

```
#include <string.h>
int strcoll(const char *s1, const char *s2);
```

说明

函数 strcoll 对 s1 指向的串和 s2 指向的串进行比较，只是比较时串都被解释为适合当前区域设置的类别 LC_COLLATE 的形式。

返回值

当两个串都被解释为适合当前区域设置的形式后，s1 指向的串大于、等于或者小于 s2 指向的串时，函数 strcoll 分别返回一个大于、等于或者小于零的整数。

strncmp

7.11.4.4 函数 strncmp

概述

```
#include <string.h>
int strncmp(const char *s1, const char *s2, size_t n);
```

说明

函数 strncmp 对 s1 和 s2 指向的数组中的最多 n 个字符（空字符后面的字符不参加比较）进行比较。

返回值

当 s1 指向的可能以空字符结束的数组大于、等于或者小于 s2 指向的可能以空字符结束的数组时，函数 strncmp 分别返回一个大于、等于或者小于零的整数。

strxfrm

7.11.4.5 函数 strxfrm

概述

```
#include <string.h>
size_t strxfrm(char *s1, const char *s2, size_t n);
```

说明

函数 strxfrm 转换 s2 指向的串，并把结果串复制到 s1 指向的数组中。这个转换满足，使用 strcmp 对两个转换后的串进行比较时，返回值大于、等于或者小于零，分别对应使用 strcoll 对两个相同的原始串进行比较的结果。可以把最多 n 个字符放到 s1 指向的结果数组中，包括终止的空字符。如果 n 是零，允许 s1 是一个空指针。如果复制发生在两个重叠的对象中，则行为未定义。

返回值

函数 strxfrm 返回转换串（不包括终止的空字符）的长度。如果返回的值是 n 或者更大，则 s1 指向的数组的内容是不确定的。

例子

下面的表达式的值是保存 s 指向的转换串所需要的数组大小。

```
1 + strxfrm(NULL, s, 0)
```

7.11.5 查找函数

memchr

7.11.5.1 函数 memchr

概述

```
#include <string.h>
void *memchr(const void *s, int c, size_t n);
```

说明

函数 memchr 确定 c（转换为 unsigned char 类型）在 s 指向的对象的前 n 个字符（每一个都解释为 unsigned char 类型）中第一次出现的位置。

返回值

函数返回指向定位的字符的指针，或者如果该字符没有出现在对象中则返回空指针。

strchr

7.11.5.2 函数 strchr

概述

```
#include <string.h>
char *strchr(const char *s, int c);
```

说明

函数 strchr 确定 c（转换为 char 类型）在 s 指向的串中第一次出现的位置。终止的空字符被认为是串的一部分。

返回值

函数 strchr 返回指向定位的字符的指针，或者如果该字符没有出现在对象中则返回空指针。

strcspn

7.11.5.3 函数 strcspn

概述

```
#include <string.h>
size_t strcspn(const char *s1, const char *s2);
```

说明

函数 strcspn 计算 s1 指向的字符串中完全由不是 s2 指向的串中的字符组成的最大初始段的长度。

返回值

函数 strcspn 返回段的长度。

strpbrk

7.11.5.4 函数 strpbrk

概述

```
#include <string.h>
char *strpbrk(const char *s1, const char *s2);
```

说明

函数 strpbrk 确定 s2 指向的串中的任意字符在 s1 指向的串中第一次出现的位置。

返回值

函数 strpbrk 返回指向找到的字符的指针，或者如果 s2 中没有字符出现在 s1 中，则返回空指针。

strrchr

7.11.5.5 函数 strrchr

概述

```
#include <string.h>
char *strrchr(const char *s, int c);
```

说明

函数 strrchr 确定 c（转换为 char 类型）在 s 指向的串中最后一次出现的位置。终止的空字符被认为是串的一部分。

返回值

函数 strrchr 返回指向找到的字符的指针，或者如果该字符没有出现在串中，则返回空指针。

strspn

7.11.5.6 函数 strspn

概述

```
#include <string.h>
size_t strspn(const char *s1, const char *s2);
```

说明

函数 strspn 计算 s1 指向的字符串中完全由 s2 指向的串中的字符组成的最大初始段的长度。

返回值

函数 strspn 返回段的长度。

strstr

7.11.5.7 函数 strstr

概述

```
#include <string.h>
char *strstr(const char *s1, const char *s2);
```

说明

函数 strstr 确定 s2 指向的串的字符序列（不包括终止的空字符）在 s1 指向的串中第一次出现的位置。

返回值

函数 strstr 返回指向定位串的指针，或者如果没有找到该串，则返回一个空指针。如果 s2 指向一个长度为零的串，则函数返回 s1。

strtok

7.11.5.8 函数 strtok

概述

```
#include <string.h>
char *strtok(char *s1, const char *s2);
```

说明

对函数 strtok 的连续调用把 s1 指向的串分解为一系列记号，每个记号都由 s2 指向的串中的字符界定。对这个序列的第一次调用把 s1 作为它的第一个参数，后面的调用把空指针作为它们的第一个参数。s2 指向的分隔串可能在每次调用时都不相同。

对这个序列的第一次调用是从 s1 指向的串中搜索第一个在 s2 指向的当前分隔串中没有包含的字符。如果没有找到这样的字符，那么 s1 指向的串中就没有任何记号，函数 strtok 返回一个空指针。如果找到了这样的字符，它就是第一个记号的首字符。

然后函数 strtok 就从上面找到的字符开始搜索当前分隔串中包含的字符。如果没有找到这样的字符，当前的记号就延伸到 s1 指向的串的末尾，后面对一个记号的搜索就会返回一个空指针。如果找到了这样的字符，它就会被一个空字符覆盖，这个空字符终止当前的记号。函数 strtok 保存指向下一个字符的指针，下一次对某个记号的搜索就会从这里开始。

后面的每一次调用都会用一个空指针作为第一个参数的值，并从保存的指针处开始搜索，搜索行为和上文描述的相同。

实现的行为不受调用函数 strtok 的影响。

返回值

函数 strtok 返回指向一个记号的第一个字符的指针，如果没有记号，就返回一个空指针。

例子

```
#include <string.h>
 static char str[] = "?a???b,,,#c";
char *t;

t = strtok(str, "?");  /* t points to the token "a" */
t = strtok(NULL, ","); /* t points to the token "??b" */
t = strtok(NULL, "#,");/* t points to the token "c" */
t = strtok(NULL, "?"); /* t is a null pointer */
```

7.11.6 其他函数

memset

7.11.6.1 函数 memset

概述

```
#include <string.h>
void *memset (void *s, int c, size_t n);
```

说明

函数 memset 把 c（转换为 unsigned char 类型）的值复制到 s 指向的对象的前 n 个字符的每个字符中。

返回值

函数 memset 返回 s 的值。

strerror

7.11.6.2 函数 strerror

概述

```
#include <string.h>
char *strerror(int errnum);
```

说明

函数 strerror 将 errnum 中的错误编号对应到一个错误信息串。

实现的行为不受调用函数 strerror 的影响。

返回值

函数 strerror 返回指向一个串的指针，这个串的内容是由实现定义的。指向的数组不能由程序修改，但可能通过对函数 strerror 的后续调用而被覆盖。

strlen

7.11.6.3 函数 strlen

概述

```
#include <string.h>
size_t strlen(const char *s);
```

说明

函数 strlen 计算 s 指向的串的长度。

返回值

函数 strlen 返回终止的空字符前面的字符的数目。

脚注

133. 参考“库的展望”(7.13.8)。
134. 因此，如果 s2 指向的前 n 个字符中没有空字符，结果就不是以空字符结尾的。
135. 因此，s1 指向的数组的字符的最大数目是 strlen (s1)+n+1。
136. 结构对象内部为了对齐而填充的“空隙”的内容是不确定的。相比之下，比分配的空间短的串和联合也会造成问题。

14.3 <string.h> 的使用

可以使用 <string.h> 中的函数对字符串进行操作。可以使用指向串的起始位置的指针参数（可以记为 s）来描述串的特征。

- 如果一个串可以包含空字符，那么也必须指定它的长度（记为 n）作为一个附加的参数。n 可以为零。此时使用名字以 mem 开头的函数。
- 如果一个串可能有，也可能没有终止的空字符，那么也必须指定它的最大长度 n，n 可以为零。此时使用名字以 strn 开头的函数。
- 如果一个串一定以空字符结束，那么只需指定 s。此时使用其余的那些名字以 str 开头的函数。

除了这个简单的分类，其他串函数之间的联系就很小了。除了 <string.h> 中定义的宏和类型，下面还会单独地对每一个函数进行描述。

NULL NULL——参考 11.3 节。

size_t size_t——参考 11.3 节。

memchr memchr——使用这个函数来确定一个字符在一个已知长度的字符序列中第一次出现的位置（下标最小的那一个）。这个函数把第一个参数（字符串指针）强制转换为指向 *unsigned char* 类型的指针，把第二个参数（要搜索的字符）强制转换为 *unsigned char* 类型。这样可以保证任何字符类型的参数表达式行为都很合理并且可以预见。然而，若搜索失败则返回空指针。所以在使用返回值来访问内存之前一定要对它进行测试。同样也要注意返回值是指向 *void* 类型的指针。可以把这个值赋给一个字符指针，但是不能用它来访问内存，除非提前把它强制转换为某种字符指针类型。

memcmp memcmp——这个函数提供了确定两个相等长度的字符序列的字符是否一一对应的最快的方式。也可以使用它确定两个字符序列的词典顺序，但是这种顺序在不同的实现下可能不同。如果结果的可移植性很重要，就必须单独编写对应的比较函数。

memcpy memcpy——如果确定目标串 s1 和源串 s2 不重叠，memcpy(s1,s2,n) 将快速而安全地执行复制。如果它们两个可能重叠，可以使用 memmove(s1,s2,n) 来代替它。不要认为这两个函数以任何一种特殊的顺序访问存储空间。特别地，如果想在一个字符数组的相邻元素构成的序列中存储相同的值，可以使用 memset。

memmove memmove——参考上面的 memcpy。

memset memset——这是在字符数组中的相邻元素构成的序列中存储相同的值的安全方式。

strcat strcat——如果只需要连接两个串 s1 和 s2 或者几个短串，使用 strcat(s1,s2)。否则，可以使用例如 strcpy(s1+=strlen(s1),s2) 这样的形式。这样可以避免对串的初始部分进行重复的和不断延长的重扫描。要保证目标数组足够大可以存放连接起来的串。注意，strcat 返回 s1，而不是指向串的新的尾部的指针。

strchr strchr——使用这个函数确定一个字符在一个以空字符结尾的字符串中第一次出现的位置（下标最小的那个）。函数把第二个参数（搜索的字符）强制转换为 char 类型，这样可以保证任意字符类型的参数表达式的行为都很合理并且可以预见。然而，若搜索失败，则返回空指针。所以，在想要使用返回值访问存储空间之前，要保证先对它进行测试。注意对 strchr(s,'\0') 的调用返回一个指向终止的空字符的指针。也可以参考下面讲述的 strcspn、strpbrk 和 strrchr。

strcmp strcmp——这个函数提供了确定两个以空字符结尾的字符串的字符是否一一对应的最快的方式。也可以使用它确定两个串的词典顺序，但是这种顺序在不同的实现下可能不同。如果结果的可移植性很重要，则必须编写对应的比较函数。也可以参考下面描述的 strcoll 和 strxfrm。

strcoll strcoll——使用这个函数来确定两个以空字符结尾的字符串在指定区域设置下的词典顺序。用户必须知道当前的区域设置类别 LC_COLLATE 的状态，这样才可以妥当地使用这个函数。（至少必须假设其他某个人已经合理地设置了这个类别。）在某些环境下，可能会使用下面讲述的 strxfrm 来代替它。

strcpy strcpy——如果可以保证目标串 s1 和源串 s2 不重叠，那么 strcpy(s1,s2) 将安全并快速地执行复制操作。如果它们可能重叠，使用 memmove(s1,s2,strlen(s2)+1) 来代替它。不要认为这两个函数可以以任何特定的顺序访问内存。

strcspn strcspn——strcspn 和 strchr 很相似，但它匹配的是任意一个字符集而不是一个字符。这使得它也和 strpbrk 很相似。然而，注意，strcspn 返回一个串的索引而不是指向其中一个元素的指针。如果它没有发现匹配，它返回终止的空字符的索引，而不是一个空指针。因此，调用 strcspn(s,"a") 会比调用 strchr(s,'a') 或者 strpbrk(s,"a") 更加方便。

strerror strerror——使用 strerror（errcode）来确定和错误编码 errcode 相对应的以空字符结尾的信息串。（第 3 章描述了宏 errno 和标准错误编码。）errcode 应该是 errno 或者 <errno.h> 中定义的名字以 E 开头的某个宏。要保证在下一次调用 strerror 之前复制或者写出这些信息。如果只是想把包含 strerror(errno) 的信息写入标准错误流，可以参考 <stdio.h> 中声明的 perror。

strlen strlen——在任何可能的地方使用这个函数来确定一个以空字符结尾的字符串的长度。它可能用内联代码实现。

strncat strncat——strncat(s1,s2,n2)中的 strn 是指函数连接到以空字符结尾的串 s1 的尾部的串 s2。函数最多复制 n2 个字符，如果它没有复制终止的空字符的话，还会加上一个终止的空字符。因此，调用 strncat 的结果最多使 strlen(s1) 增大 n2。这就使 strncat 比 strcat 更加安全，虽然可能以截取 s2 的前 n2 个字符为代价。

strncmp strncmp——这个函数提供了比较两个具有相同长度的字符序列是否一一对应的最快的方式，最多可以比较到这两个串中的任意空字符而且包含该空字符。也可以使用它来对两个这样的字符序列进行词典排序，但是这种顺序依实现而不同。如果结果的可移植性很重要，那么就必须编写自己的比较函数。

strncpy strncpy——如果可以确定目标串 s1 和源串 s2 没有重叠，strncpy(s1,s2,n2) 将安全地执行复制操作。然而，函数在 s1 处精确地存储 n2 个字符，它可能丢弃尾部的字符，包括终止的空字符。函数会根据需要存储一些额外的空字符来弥补一个短的计数。如果两个区域可能重叠，使用 memmove(s1,s2,n2) 来代替它。（必须在结尾存储适当数量的空字符，如果这很重要的话。）不要认为这两个函数以任意特定的顺序访问存储空间。

strpbrk strpbrk——strpbrk 和 strchr 很相同，不过它匹配的是任意一个字符集而不是一个字符。这使得它也和 strcspn 很相似。然而，注意，strcspn 返回一个串的索引而不是指向其中一个元素的指针。如果它没有发现匹配，它返回终止的空字符的索引而不是一个空指针。因此，调用 strcspn(s,"abc") 会比调用 strpbrk(s,"abc") 更加方便。

strrchr strrchr——使用这个函数来确定一个字符在一个以空字符结尾的字符串中最后一次出现的位置（下标最大的一个）。函数把第二个参数（搜索字符）强制转换为 char 类型。这样可以保证任何字符类型的参数表达式的行为都很合理并可预见。然而，若搜索失败，则返回空指针。所以，在想要使用返回值访问存储空间之前，要保证先对它进行测试。注意，strrchr(s,'\0') 返回一个指向终止的空字符的指针。也可以参考上面讲述的 strchr、strcspn 和 strpbrk。

strspn strspn——可以把 strspn 看作是 strcspn 的补函数。它搜索一个和一个字符集中的任意元素都不匹配的字符，而不是和任意一个匹配。strspn 也返回这个串的索引，或者如果它没有找到匹配，就返回终止的空字符的索引。因此，调用 strspn(s,"abc") 会找到来自集合 "abc" 的最长可能的字符跨越。

strstr strstr——可以用 strstr(s1,s2) 来确定子串 s2 在字符串 s1 中第一次出现的位置。一个成功的搜索返回指向 s1 中的子串的起始位置的指针。若搜索失败，则返回一个空指针。

strtok strtok——这是一个复杂的函数，它设计的目的是把一个空字符结尾的字符串分解为多个记号。可以指定分隔字符的集合。记号之间存在一个或者多个分隔符的序列。这样的序列也可以存在于第一个记号之前或者最后一个记号之后。strtok 内部记录了停止分析串的位置。因此，使用 strtok 一次只能对一个串进行处理。例如，这里有一个代码序列，它对串 line 中的每个“字”调用 word 函数。代码序列把一个字定义为不包含“空白”的最长可能的字符序列，这里的空白定义为空格符、水平制表符或者换行符。

```
char *s;

for (s = line; (s = strtok(s, " \t\n")) != NULL; s = NULL)
    word (s);
```

第一次调用 strtok 时，它的第一个参数是一个非空指针。这样就开始了从 line 的起始位置的扫描。后读调用用 NULL 代替了这个参数来继续扫描。如果任何调用的返回值都不是一个空指针，那么它指向一个不包含分隔符的且以空字符结尾的字符串。注意，strtok 把空字符存储在从 line 开始的串中。所以要保证这个存储空间是可写的，并且不需要为以后的处理进行保留。

顺便说一下，可以为处理一个给定串的 strtok 的每一次调用指定不同的分隔符集合。

strxfrm strxfrm——使用 strxfrm(s1,s2,n) 来将空字符结尾的字符串 s2 转换为 s1 处的一个(不重叠的)版本。通过这种方法转换的串以后可以通过 strcmp 来进行比较。这个比较就确定了转换前的两个串在指定区域设置中的词法顺序。必须知道区域设置类别 LC_COLLATE 的当前状态，这样才能妥当地使用这个函数。(至少必须假设其他某个人已经合理地设置了这个类别。)在大多数环境下，可能会使用上面讲过的 strcoll 来代替它。如果计划进行重复的比较或者区域设置可能在比较之前改变，就可以使用 strxfrm。使用 <stdlib.h> 中声明的函数 malloc 来为 s1 分配存储空间，就像下面的代码那样：

```
size_t n = strxfrm(NULL, s2, 0);
char *s1 = malloc (n + 1);

if (s1)
    strxfrm(s1, s2, n);
```

对 strxfrm 的第一次调用确定了需要的存储空间的大小。第二次调用（又一次）执行转换并且把转换后的串存储在分配的数组中。

14

14.4　<string.h> 的实现

<string.h> 中声明的函数工作时彼此都很独立。例外的函数只有 strcoll 和 strxfrm。它们通过两种不同的方式执行本质上相同的操作，我在最后讨论它们。剩下的每一个函数都执行一个相当简单的操作。这里的目标是把它们编写得清晰、健壮并且高效。

头文件 <string.h>

图 14-1 显示了文件 string.h。和往常一样，它从内部头文件 <yvals.h> 中继承了那些在几个标准头文件中重复使用的定义。第 11 章已经讨论了宏 NULL 和类型定义 size_t 的实现。

图 14-1 string.h

```
/* string.h standard header */
#ifndef _STRING
#define _STRING
#ifndef _YVALS
#include <yvals.h>
#endif

        /* macros */
#define NULL    _NULL
        /* type definitions */
#ifndef _SIZET
#define _SIZET
typedef _Sizet size_t;
#endif
        /* declarations */
void *memchr(const void *, int , size_t);
int memcmp(const void *, const void *, size_t);
void *memcpy(void *, const void *, size_t);
void *memmove(void *, const void *, size_t);
void *memset(void *, int, size_t);
char *strcat(char *, const char *);
char *strchr(const char *, int);
int strcmp(const char *, const char *);
int strcoll(const char *, const char *);
char *strcpy(char *, const char *);
size_t strcspn(const char *, const char *);
char *strerror(int);
size_t strlen(const char *);
char *strncat(char *, const char *, size_t);
int strncmp(const char *, const char *, size_t);
char *strncpy(char *, const char *, size_t);
char *strpbrk(const char *, const char *);
char *strrchr(const char *, int);
size_t strspn(const char *, const char *);
char *strstr(const char *, const char *);
char *strtok(char *, const char *);
size_t strxfrm(char *, const char *, size_t);
char *_Strerror(int , char *);
        /* macro overrides */
#define strerror(errcode) _Strerror(errcode, _NULL)
#endif
```
□

只有函数 strerror 使用了一个屏蔽宏。它和 <stdio.h> 中声明的函数 perror 共用内部函数 _Strerror，12.4 节讨论了其中的原因。

<string.h> 中声明的其他几个函数可以作为生成内联代码的内置函数实现。通常的做法是用私有的名字给这些内置版本命名，然后可以提供屏蔽宏来获得对这些内置函数的访问。（参考 0.2 节的 C 标准的脚注 96。）因此，<string.h> 的产品版本可能会包含几个附加的屏蔽宏。

函数 memchr

让我们从 mem 函数开始。图 14-2 显示了文件 memchr.c。函数 memchr 的主要关心的是分清各种各样的类型。必须把指针和字符参数赋值给不同类型的动态数据对象。这就需要把数组元素作为 *unsigned char* 类型正确而高效地进行比较。返回值表达式中编写 (void*) 强制类型转换是为了清晰性，它并不是必需的。

函数 memcmp

图 14-3 显示了文件 memcmp.c。memcmp 也要认真地执行 *unsigned char* 类型的比较来满足 C 标准的要求。

函数 memcpy

图 14-4 显示了文件 memcpy。我选择 *char* 作为 memcpy 的内部工作类型，某些计算机体系结构可能会使用 *unigned char*，但这种可能性很小。（这是选择一个“普通的”字符类型的一个理由。）memcpy 可以假设它的源串和

图 14-2
memchr.c

```
/* memchr function */
#include <string.h>

void *(memchr)(const void *s, int c, size_t n)
    {                          /* find first occurrence of c in s[n] */
    const unsigned char uc = c;
    const unsigned char *su;

    for (su = s; 0 < n; ++su, --n)
        if (*su == uc)
            return ((void *)su);
    return (NULL);
    }
```

图 14-3
memcmp.c

```
/* memcmp function */
#include <string.h>

int (memcmp)(const void *s1, const void *s2,
    size_t n)
    {                          /* compare unsigned char s1[n], s2[n] */
    const unsigned char *su1, *su2;

    for (su1 = s1, su2 = s2; 0 < n; ++su1, ++su2, --n)
        if (*su1 != *su2)
            return ((*su1 < *su2) ? -1 : +1);
    return (0);
    }
```

14

图 14-4 memcpy.c

```
/* memcpy function */
#include <string.h>

void *(memcpy)(void *s1, const void *s2, size_t n)
    {                       /* copy char s2[n] to s1[n] in any order */
    char *su1;
    const char *su2;

    for (su1 = s1, su2 = s2; 0 < n; ++su1, ++su2, --n)
        *su1 = *su2;
    return (s1);
    }
```

图 14-5 memmove.c

```
/* memmove function */
#include <string.h>

void *(memmove)(void *s1, const void *s2, size_t n)
    {                           /* copy char s2[n] to s1 [n] safely */
    char *sc1;
    const char *sc2;

    sc1 = s1;
    sc2 = s2;
    if (sc2 < sc1 && sc1 < sc2 + n)
        for (sc1 += n, sc2 += n; 0 < n; --n)
            *--sc1 = *--sc2;                        /*copy backwards */
    else
        for (; 0 < n; --n)
            *sc1++ = *sc2++;                        /* copy forwards */
    return (s1);
    }
```

图 14-6 memset.c

```
/* memset function */
#include <string.h>

void *(memset) (void *s, int c, size_t n)
    {                       /* store c throughout unsigned char s[n] */
    const unsigned char uc = c;
    unsigned char *su;

    for (su = s; 0 < n; ++su, --n)
        *su = uc;
    return (s);
    }
```

目的串不重叠。因此，它可以执行它能做的最简单的复制操作。

函数 memmove 图 14-5 显示了文件 memmove.c。函数 memmove 必须能正确地工作，甚至当它的两个操作数有重叠的时候。因此，它首先检查一个可能会阻止一个升序复制的正确操作的重叠。如果有重叠发生，它就按照降序复制元素。

图 14-7 strncat.c

```
/* strncat function */
#include <string.h>

char *(strncat)(char *s1, const char *s2, size_t n)
    {                       /* copy char s2[max n] to end of s1[] */
    char *s;

    for (s = s1; *s ! = '\0'; ++s)
        ;                                       /* find end of s1[] */
    for (; 0 < n && *s2 ! = '\0'; --n)
        *s++ = *s2++;               /* copy at most n chars from s2[] */
    *s = '\0';
    return (s1);
    }
```

图 14-8 strncmp.c

```
/* strncmp function */
#include <string.h>

int (strncmp)(const char *s1, const char *s2, size_t n)
    {               /* compare unsigned char s1[max n] , s2 [max n] */
    for (; 0 < n; ++s1, ++s2, --n)
        if (*s1 ! = *s2)
            return ((*(unsigned char *)s1
                < *(unsigned char *)s2) ? -1 : +1);
        else if (*s1 == '\0')
            return (0);
    return (0);
    }
```

函数 memset　图 14-6 显示了文件 memset.c。我选择 *unsigned char* 作为 memset 内部的工作类型，某个实现可能会产生一个把 *int* 值存储在其他字符类型中的溢出，但是这种可能性极小。

函数 strncat　现在考虑一下 3 个 strn 函数，图 14-7 显示了文件 strncat.c。函数 strncat 首先确定目标串的终点位置，然后，它把源串中的最多 n 个字符连接上去。注意这个函数总是提供一个终止的空字符。

函数 strncmp　图 14-8 显示了文件 strncmp.c。函数 strncmp 和 memcmp 很相似，但是它也在一个终止的空字符处停止。而且和 memcmp 不一样的是，strncmp 可以直接使用它的指针参数。它可以把它们强制转换为指向 *unsigned char* 类型的指针，这仅仅为了计算一个非零的返回值。

函数 strncpy　图 14-9 显示了文件 strncpy.c。同样，函数 strncpy 和 memcpy 很相似，只不过它在一个终止的空字符处停止。strncpy 也有个不恰当的要求，就是对一个长度小于 n 的串来说，它必须用空字符来填充。

strcat strcmp strcpy　str 函数中有 3 个和 strn 函数非常相似。图 14-10 到图 14-12 显示了文件 strcat.c、strcmp.c 和 strcpy.c。函数 strcat、strcmp 和 strcpy 与 strn 函数的不同之处只在于不用担心有限的串长度 n。当然，strcpy 也不用处理填充字符。

图 14-9
strncpy.c

```
/* strncpy function */
#include <string.h>

char * (strncpy)(char *s1, const char *s2, size_t n)
    {                               /* copy char s2 [max n] to s1[n] */
    char *s;

    for (s = s1; 0 < n && *s2 != '\0'; --n)
        *s++ = *s2++;               /* copy at most n chars from s2[] */
    for (; 0 < n; --n)
        *s++ = '\0';
    return (s1);
    }
```

图 14-10
strcat.c

```
/* strcat function */
#include <string.h>

char *(strcat)(char *s1, const char *s2)
    {                               /* copy char s2[] to end of s1[] */
    char *s;

    for (s = s1; *s != '\0'; ++s)
        ;                                   /* find end of s1[] */
    for (;(*s = *s2) != '\0'; ++s, ++s2)
        ;                                   /* copy s2[] to end */
    return (s1);
    }
```

图 14-11
strcmp.c

```
/* strcmp function */
#include <string.h>

int(strcmp)(const char *s1,
    const char *s2)
    {                       /* compare unsigned char s1[], s2[] */
    for (; *s1 == *s2; ++s1, ++s2)
        if(*s1 == '\0')
            return (0);
    return ((*(unsigned char *) s1
        < *(unsigned char *)s2) ? -1 : +1);
    }
```

图 14-12
strcpy.c

```
/* strcpy function */
#include <string.h>

char *(strcpy)(char *s1, const char *s2)
    {                                   /* copy char s2[] to s1[] */
    char *s = s1;

    for (s = s1; (*s++ = *s2++) != '\0';)
        ;
    return (s1);
    }
```

图 14-13
strlen.c

```
/* strlen function */
#include <string.h>

size_t (strlen)(const char *s)
    {                                       /* find length of s[] */
    const char *sc;

    for (sc = s; *sc != '\0'; ++sc)
        ;
    return (sc - s);
    }
```

图 14-14
strchr.c

```
/* strchr function */
#include <string.h>

char *(strchr)(const char *s, int c)
    {                   /* find first occurrence of c in char s[] */
    const char ch = c;

    for (; *s != ch; ++s)
        if (*s == '\0')
            return (NULL);
    return ((char *) s);
    }
```

图 14-15
strcspn.c

```
/* strcspn function */
#include <string.h>

size_t (strcspn)(const char *s1, const char *s2)
    {           /* find index of first s1[i] that matches any s2[] */
    const char *sc1, *sc2;

    for (sc1 = s1; *sc1 != '\0'; ++sc1)
        for (sc2 = s2; *sc2 != '\0'; ++sc2)
            if (*sc1 == *sc2)
                return (sc1 - s1);
    return (sc1 - s1);                      /* terminating nulls match */
    }
```

函数 strlen 图 14-13 显示了文件 strlen.c。函数 strlen 可能是 <string.h> 中声明的函数中使用最频繁的一个。它最有可能作为内置函数实现。如果这种形式存在，就去寻找 strlen 隐藏为内联代码的地方。函数 strcat 和 strncat 就是两个明显的例子。

函数 strchr 有 7 个函数通过各种不同的方式扫描字符串。图 14-14 显示了文件 strchr.c。函数 strchr 是这些函数中最简单的一个，它明显和 memchr 很相似。

strcspn
strpbrk
strspn

图 14-15 到图 14-17 显示了文件 strcspn.c、strpbrk.c 和 strspn.c。strcspn 和 strpbrk 执行相同的功能，只是返回值不同。函数 strspn 是 strcspn 的补函数。

图 14-16
strpbrk.c

```
/* strpbrk function */
#include <string.h>

char *(strpbrk)(const char *s1, const char *s2)
    {          /* find index of first s1[i] that matches any s2[] */
    const char *sc1, *sc2;

    for (sc1 = s1; *sc1 != '\0'; ++sc1)
        for (sc2 = s2; *sc2 != '\0'; ++sc2)
            if (*sc1 == *sc2)
                return ((char *)sc1);
    return (NULL);                      /* terminating nulls match */
    }
```

图 14-17
strspn.c

```
/* strspn function */
#include <string.h>

size_t (strspn)(const char *s1, const char *s2)
   {            /* find index of first s1[i] that matches no s2[] */
    const char *sc1, *sc2;

    for (sc1 = s1; *sc1 != '\0'; ++sc1)
        for (sc2 = s2; ; ++sc2)
            if (*sc2 == '\0')
                return (sc1 - s1);
            else if (*sc1 == *sc2)
                break ;
    return (sc1 - s1);                        /* null doesn't match */
    }
```

图 14-18
strrchr.c

```
/* strrchr function */
#include <string.h>

char *(strrchr)(const char *s, int c)
    {                      /* find last occurrence of c in char s[] */
    const char ch = c;
    const char *sc;

    for (sc = NULL; ; ++s)
        {                                     /* check another char */
        if (*s == ch)
            sc = s;
        if (*s == '\0')
            return ((char *)sc);
        }
    }
```

函数 strrchr　图 14-18 显示了文件 strrchr.c。函数 strrchr 是对 strchr 的一个有用的补。它记录指向 sc 中的最右边的元素（如果有的话）的指针。返回语句中的强制类型转换是必需的，因为在这种情况下，sc 指向一个常量类型。

图 14-19 strstr.c

```
/* strstr function */
#include <string.h>

char *(strstr)(const char *s1, const char *s2)
	{                   /* find first occurrence of s2[] in s1[] */
	if (*s2 == '\0')
		return ((char *)s1);
	for (; (s1 = strchr(s1, *s2)) != NULL; ++s1)
		{                                /* match rest of prefix */
		const char *sc1, *sc2;

		for (sc1 = s1, sc2 = s2; ; )
			if (*++sc2 == '\0')
				return ((char *)s1);
			else if (*++sc1 != *sc2)
				break;
		}
	return (NULL);
	}
```

图 14-20 strtok.c

```
/* strtok function */
#include <string.h>

char *(strtok)(char *s1, const char *s2)
	{                 /* find next token in s1[] delimited by s2[] */
	char *sbegin, *send;
	static char *ssave = "";                       /* for safety */

	sbegin = s1 ? s1 : ssave;
	sbegin += strspn(sbegin, s2);
	if (*sbegin == '\0')
		{                                          /* end of scan */
		ssave = "";                                /* for safety */
		return (NULL);
		}
	send = sbegin + strcspn(sbegin, s2);
	if (*send != '\0')
		*send++ = '\0';
	ssave = send;
	return (sbegin);
	}
```

函数 strstr 图 14-19 显示了文件 strstr.c。函数 strstr 调用 strchr 来在 s1 中寻找串 s2 的第一个字符。只有找到时，它才使用所需的东西来检查 s2 的其他部分是否和 s1 的一个子串相匹配。这个函数把空串 s2 作为一种特殊情况对待，它和 s1 起始位置隐藏的空串相匹配。

函数 strtok 图 14-20 显示了文件 strtok.c。函数 strtok 是 7 个串扫描函数中最后一个也是最复杂的一个。它看起来并不是很糟因为这里它是在 strspn 和 strpbrk 的基础上编写出来的。然而，它必须对付可写的静态存储空间和多

重调用问题来处理相同的串。可能正确地使用它至少和正确地编写它一样难。当 strtok 没有被激活来扫描一个参数串时，它指向一个空串。这至少可以阻止不正当的调用导致函数对存储空间的无效访问。（如果它正在扫描的串的存储空间被释放了，这个函数仍然很危险。）

strerror
_Strerror

图 14-21 显示了文件 strerror.c。它既定义了 strerror，也定义了内部函数 _Strerror。（参考 12.4 节中对 <stdio.h> 中声明的函数 perror 调用 _Strerror 的原因的说明。）_Strerror 在一个缓冲区中建立某些错误编码的文本表示，它只有被 strerror 调用的时候才使用它自己的静态缓冲区。这里我只对 <errno.h> 的这个实现中定义的错误编码的最小集合提供具体的信息。用户可能会想添加更多。任何未知的错误编码都作为 3 位 10 进制数打印输出。

图 14-21
strerror.c

```
/* strerror function */
#include <errno.h>
#include <string.h>

char *_Strerror(int errcode, char *buf)
    {                    /* copy error message into buffer as needed */
    static char sbuf[] = {"error #xxx"};

    if (buf == NULL)
        buf = sbuf;
    switch (errcode)
        {                                /* switch on known error codes */
    case 0:
        return ("no error");
    case EDOM:
        return ("domain error");
    case ERANGE:
        return ("range error");
    case EFPOS :
        return ("file positioning error");
    default:
        if (errcode < 0 || _NERR <= errcode)
            return ("unknown error");
        else
            {                            /* generate numeric error code */
            strcpy (buf, "error #xxx");
            buf[9] = errcode % 10 + '0';
            buf[8] = (errcode /= 10) % 10 + '0';
            buf[7] = (errcode / 10) % 10 + '0';
            return (buf);
            }
        }
    }

char *(strerror)(int errcode)
    {                /* find error message corresponding to errcode */
    return (_Strerror(errcode, NULL));
    }
```
□

整理函数

<string.h> 中声明的最后两个函数帮助执行特定区域设置下的字符串整理。strcoll 和 strxfrm 都通过把串转换为某种形式来确定整理顺序，这种形式使用 strcmp 比较时，可以得到正确的整理。区域设置类别 LC_COLLATE 确定这种转换。（参考第 6 章。）它通过指定内部函数 _Strxfrm 使用的状态表来完成这个功能。因此 strcoll 和 strxfrm 调用 _Strxfrm 来正确地转换字符串。

头文件 "xstrxfrm.h"

图 14-22 显示了文件 xstrxfrm.h。所有的整理函数都包含内部头文件 "xstrxfrm.h"。该头文件又包含标准头文件 <string.h> 和内部头文件 "xstate.h"。（参考图 6-5 的文件 xstate.h。）除此之外，"xstrxfrm.h" 定义了类型 _Cosave 并且声明了函数 _Strxfrm。一个 _Cosave 类型的数据对象在调用 _Strxfrm 之间存储状态信息。

函数 strxfrm

图 14-23 显示了文件 strxfrm.c。函数 strxfrm 最好地阐述了整理函数是怎样在一起工作的。它在 s1 指向的缓冲区中存储了长度为 n 的转换字符串。一旦缓冲区满，函数就转换源串的剩余部分来确定转换串的总长度。strxfrm 把所有多余的字符存储在它自己的动态临时缓冲区 buf 中。

函数 _Strxfrm

图 14-24 显示了文件 xstrxfrm.c，它定义了函数 _Strxfrm，这个函数执行了实际的转换。它是作为执行存储在 _Wcstate 中的状态表的有限状态机来实现这个功能的，_Wcstate 由文件 xstate.c 定义。（参考图 6-11。）

_Strxfrm 一定要特别严密，因为 _Wcstate 可能会有缺陷。它可能会随着区域类别 LC_COLLATE 而改变，且改变方式是 C 标准库不能控制的。

注意，出现以下情况时，函数可以选择错误返回。

- 如果发生了向一个未定义的状态的转移。
- 如果一个给定的状态不存在对应的状态表。
- 如果函数产生一个输出字符时要做太多的状态转移以致它必须要循环。
- 如果状态表入口专门报告了一个错误。

图 14-22 xstrxfrm.h

```
/* xstrxfrm.h internal header */
#include <string.h>
#include "xstate. h"
        /* type definitions */
typedef struct {
    unsigned char _State;
    unsigned short _Wchar;
    } _Cosave;
```

图 14-23 strxfrm.c

```
/* strxfrm function */
#include "xstrxfrm.h"

size_t (strxfrm)(char *s1, const char *s2, size_t n)
    {           /* transform s2[] to s1[] by locale-dependent rule */
    size_t nx = 0;
    const unsigned char *s = (const unsigned char *)s2;
    _Cosave state = {0};

    while (nx < n)
        {                                /* translate and deliver */
        size_t i = _Strxfrm(s1, &s, n - nx, &state);

        s1 += i, nx += i;
        if (0 < i && s1[-1] == '\0')
            return (nx - 1);
        else if (*s == '\0')
            s = (const unsigned char *)s2;              /* rescan */
        }
    for (; ; )
        {                                  /* translate and count */
        char buf[32];
        size_t i = _Strxfrm(buf, &s, sizeof (buf), &state);

        nx += i;
        if (0 < i && buf[i - 1] == '\0')
            return (nx - 1);
        else if (*s == '\0')
            s = (const unsigned char *)s2;              /* rescan */
        }
    }
```

相比较而言，_Strxfrm 的剩余部分都是简单的。这个函数把宽字节字符累加器 (ps->_Wchar) 作为状态存储的一部分保留下来。在一个给定的转移状态下时，这就简化了用一个普通的组件生成一个转换字符序列的工作。_Strxfrm 在它（用 size 个字符）填满输出缓冲后，或者遇到源串中的终止的空字符时返回。

这可能会不只一次发生。注意 strxfrm 辨别 _Strcfrm 返回的 3 种原因的方式：

- 如果传递的最后一个字符是空字符，转换就完成了。如果发生了一个错误，_Strxfrm 就传递一个空字符。它也会记录存储的状态信息，这样，当它再次不小心调用同样的串时，它就会立即失败。
- 否则，如果下一个源字符是空字符，_Strxfrm 会重新扫描源串。_Strxfrm 不会越过源串中的一个空字符。
- 否则，_Strxfrm 会在它停止的地方继续执行。

图 14-24
xstrxfrm.c

```
/* _Strxfrm function */
#include <limits.h>
#include "xstrxfrm.h"

size_t _Strxfrm(char *sout, const unsigned char **psin,
    size_t size, _Cosave *ps)
    {                       /* translate string to collatable form */
    char state = ps->_State;
    int leave = 0;
    int limit = 0;
    int nout = 0;
    const unsigned char *sin = *psin;
    unsigned short wc = ps->_Wchar;

    for (; ; )
        {                       /* perform a state transformation */
        unsigned short code;
        const unsigned short *stab;

        if (_NSTATE <= state
            || (stab = _Costate._Tab[state]) == NULL
            || (_NSTATE*UCHAR_MAX) <= ++limit
            || (code = stab[*sin]) == 0)
            break;
        state = (code & ST_STATE) >> ST_STOFF;
        if (code & ST_FOLD)
            wc = wc & ~UCHAR_MAX | code & ST_CH;
        if (code & ST_ROTATE)
            wc = wc >> CHAR_BIT & UCHAR_MAX | wc << CHAR_BIT;
        if (code & ST_OUTPUT && ((sout[nout++]
            = code & ST_CH ? code : wc) == '\0'
            || size <= nout))
            leave = 1;
        if (code & ST_INPUT)
            if (*sin != '\0')
                ++sin, limit = 0;
            else
                leave = 1;
        if (leave)
            {                       /* return for now */
            *psin = sin;
            ps->_State = state;
            ps->_wchar = wc;
            return (nout);
            }
        }
    sout [nout++] = '\0';                       /*error return */
    *psin = sin;
    ps->_State = _NSTATE;
    return (nout);
    }
```
□

图 14-25
strcoll.c

```
/* strcoll function */
#include "xstrxfrm. h"

        /* type definitions */
typedef struct {
    char buf[32];
    const unsigned char *s1, *s2, *sout;
    _Cosave state;
    } Sct1;

static size_t getxfrm(Sct1 *p)
    {                                    /* get transformed chars */
    size_t i;

    do {                                 /* loop until chars delivered */
        p->sout = (const unsigned char *)p->buf;
        i = _Strxfrm(p->buf, &p->s1, sizeof (p->buf), &p->state);
        if (0 < i && p->buf[i - 1] == '\0')
            return (i - 1);
        else if (*p->s1 == '\0')
            p->s1 = p->s2;                                /* rescan */
        } while (i == 0);
    return (i);
    }

int (strcoll)(const char *s1, const char *s2)
    {           /* compare s1[], s2[] using locale-dependent rule */
    size_t n1, n2;
    Sct1 st1, st2;
    static const _Cosave initial = {0};

    st1.s1 = (const unsigned char *)s1;
    st1.s2 = (const unsigned char *)s1;
    st1.state = initial;
    st2.s1 = (const unsigned char *)s2;
    st2.s2 = (const unsigned char *)s2;
    st2.state = initial;
    for (n1 = n2 = 0; ; )
        {                                /* compare transformed chars */
        int ans;
        size_t n;

        if (n1 == 0)
            n1 = getxfrm(&st1);
        if (n2 == 0)
            n2 = getxfrm(&st2);
        n = n1 < n2 ? n1 : n2;
        if (n == 0)
            return (n1 == n2 ? 0 : 0 < n2 ? -1 : +1);
        else if ((ans = memcmp(st1.sout, st2.sout, n)) != 0)
            return (ans);
        st1.sout += n, n1 -= n;
        st2.sout += n, n2 -= n;
        }
    }
```
□

函数 strcoll　图 14-25 显示了文件 strcoll.c，函数 strcoll 比 strxfrm 稍微复杂点。它必须对两个源串进行转换，并且一次只能转换一段，这样它就可以比较它们的转换形式。类型 Sctl 描述了保存处理每个源串所需信息的数据对象。内部函数 getxfrm 调用 _Strxfrm 来对 Sctl 数据对象进行更新。

strcoll 内部的循环比较对 Sctl 缓冲中每一个没有转换字符的源串，都会调用 getxfrm。这就保证了如果还需要产生任何这样的字符，则每一个源串至少用一个转换字符表示。strcoll 比较它所能比较的所有转换字符。只有当两个转换串的对应字符都相等并且长度相同时，它才返回零。

14.5 <string.h> 的测试

图 14-26 显示了文件 tstring.c。测试程序对 <string.h> 中声明的每一个函数执行了一些粗略的测试。这个头文件没有定义独有的宏或者类型，所以，也没有什么有意义的大小可以显示。如果一切顺利，测试程序只显示如下信息：

```
SUCCESS testing <string.h>
```

14.6 参考文献

R.E. Griswold, J.F. Poage, and I.P. Polonsky, *The SNOBOL4 Programming Language*, (Englewood Cliffs, N.J.: Prentice-Hall, Inc. 1971). 程序设计语言 SNOBOL 把模式匹配和文本串内的替换推向了顶峰。读者可能会对在字符串操作的基础上可以完成的功能强大的程序而感到惊奇。

14.7 习题

14.1 下面的区域设置文件定义了一个简单的“字典”整理顺序，这个顺序忽略了标点符号和大小写的区别。

```
LOCALE DICT
NOTE dictionary collation sequence
collate[0, 0       ] '.'        $O $I $1
collate[0, 1:$#    ]               $I $0
collate[0, 'a':'z'] $@          $O $I $0
collate[0, 'A':'Z'] $@+'a'-'A'  $O $I $0
collate[1, 0:$#    ] $@          $O $I $1
LOCALE and
```

描述一下它执行的转换。它为什么要重新扫描？为这个转换画一个状态转换图。

14

图 14-26 tstring.c

```
/* test string functions */
#include <assert.h>
#include <errno.h>
#include <stdio.h>
#include <string.h>

int main()
    {                    /* test basic workings of string functions */
    chat s[20];
    size_t n;
    static const char abcde[] = "abcde";
    static const char abcdx[] = "abcdx";

    assert(memchr(abcde,'c',5) == &abcde[2]);
    assert(memchr(abcde,'e',4) == NULL);
    assert(memcmp(abcde, abcdx,5) ! = 0);
    assert(memcmp(abcde, abcdx,4) == 0);
        /* the following tests are interrelated */
    assert(memcpy(s, abcde, 6) == s && s[2] == 'c');
    assert(memmove(s, s + 1,3) == s);
    assert(memcmp(memmove(s, s + 1, 3), "aabce", 6));
    assert(memcmp((char *)memmove(s + 2, s, 3) - 2,
        "bcece", 6));
    assert(memset(s,'*', 10) == s && s[9] == '*');
    assert(memset(s + 2, '%', 0) == s + 2 && s[2] == '*');
    assert(strcat(memcpy(s, abcde, 6), "fg") == s);
    assert(s[6] == 'g');
    assert(strchr(abcde, 'x') == NULL);
    assert(strchr(abcde, 'c') == &abcde[2]);
    assert(strchr(abcde, '\0') == &abcde[5]);
    assert(strcmp(abcde, abcdx) ! = 0);
    assert(strcmp(abcde, "abcde") == 0);
    assert(strcoll(abcde, "abcde") == 0);
    assert(strcpy(s, abcde) == s && strcmp(s, abcde) == 0);
    assert(strcspn(abcde, "xdy") == 3);
    assert(strcspn(abcde, "xzy") == 5);
    assert(strerror(EDOM) != 0);
    assert(strlen(abcde) == 5);
    assert(strlen("") == 0);
    assert(strncat(strcpy(s, abcde), "fg", 1) == s
        && strcmp(s, "abcdef") == 0);
    assert(strncmp(abcde, "abcde", 30) == 0);
    assert(strncmp(abcde, abcdx, 30) != 0);
    assert(strncmp(abcde, abcdx, 4) == 0);
    assert(strncpy(s, abcde, 7) == s
        && memcmp(s, "abcde\0", 7) == 0);
    assert(strncpy(s, "xyz", 2) == s
        && strcmp(s, "xycde") == 0);
    assert(strpbrk(abcde, "xdy") == &abcde[3]);
    assert(strpbrk(abcde, "xzy") == NULL);
    assert(strrchr(abcde, 'x') == NULL);
    assert(strrchr(abcde, 'c') == &abcde[2]);
    assert(strcmp(strrchr ("ababa", 'b'),"ba") == 0);
    assert(strspn(abcde, "abce") == 3);
    assert(strspn(abcde, abcde) == 5);
```

图 14-26
（续）

```
assert(strstr(abcde, "xyz") == NULL);
assert(strstr(abcde, "cd") == &abcde[2]);
assert(strtok(strcpy(s, abcde), "ac") == &s[1]);
assert(strtok(NULL, "ace") == &s[3]);
assert(strtok(NULL, "ace") == NULL
    && memcmp(s, "ab\0d\0\0", 6) == 0);
n = strxfrm(NULL, abcde, 0);
if (n < sizeof (s) - 1)
    assert(strxfrm(s, abcde, n + 1) == n
        && strlen(s) == n) ;
puts("SUCCESS testing <string.h>");
return (0);
}
```

14.2 修改上题中的区域设置文件，对以 Mac 开头和以 Mc 开头的名字进行排序，这两种名字是等价的，只有当一些名字比较为相等时才将 Mac 名字排在 Mc 名字的前边。

14.3 为下面的内容写出一个精确的规格说明：

- 你的电话簿中的名字的排序方法；
- 你使用的词典中字的排序方法；
- 你使用的计算机中的排序函数对文本行进行排序的方法。

定义一个区域设置，它可以和这些整理规则中的每一个行为相匹配。它要使用多少种状态来指定每一个规则？

14.4 一个简单的计算器程序识别下面的记号：

- <stdlib.h> 中声明的函数 strtod 可接受的数（参考图 13-1）；
- 集合 [+ - * / =C] 中的运算符；
- 双引号之间的注释。

这些记号被空格、水平制表符和换行符分隔开。然而，这样的字符可以出现在注释中。

编写一个程序，这个程序可以从标准输入流读入字符，然后把它们分解为若干个记号。使用 <string.h> 中声明的函数 strtok。重新编写该函数而不使用 strtok。你更喜欢这两个版本中的哪一个？为什么？

14.5 找出 <string.h> 中没有声明的“缺失”的函数（例如 strnlen 和 memrchr）并实现它们。能把它们添加到 C 标准库中并且仍然和 C 标准保持一致吗？能把它们的声明添加到 <string.h> 并且仍然和 C 标准保持一致吗？

14.6 测试一个大代码集，确定 <string.h> 中声明的且耗费时间最多的 5 个函数。如果这些函数瞬间就可以完成，你可以使一个典型的程序的速度提高多少？如果每一个函数都加快 5 倍，你可以使一个典型程序的速度提高多少？对于你测试的速度提高最多的程序来说，哪些方面是可以用数字来比较的？

14.7 [**难**] 为上题中找出的函数编写汇编语言版本。仅仅通过改变 C 代码函数可以大大提高它的速度吗？和它们的 C 版本相比，每一个函数可以快多少？

14.8 [**很难**] 修改一个 C 编译器，使它为你在前两道题中找到的函数产生内联代码。和它们上题中的版本相比，每个函数可以快多少？

第 15 章 <time.h>

15.1 背景知识

在 UNIX 操作系统下，时间和日期计算达到了一个新的水平，有几个系统开发者是天文爱好者。他们对表示几十年内，而不仅仅是几年内的时间的需求很敏感。他们自发地使用了格林尼治平时（过去记作 GMT，现在是 UTC），而不仅仅是墙上的钟表的时间。总之，他们比大多数人更讲究在计算机上计算和表示时间。

对这方面的细节的关注已经扩展到了 C 标准库中。它的范围基本上包括了 UNIX 下任何可用 C 实现的不依赖于 UNIX 特性的东西。因此，可以用标准 C 的时间和日期进行多种操作，`<time.h>` 中声明的函数提供了相关的服务。

说这些函数不依靠 UNIX 的特性有一点夸张。并不是所有的操作系统都区分本地时间和 UTC，甚至没什么系统允许不同的用户显示不同时区的时间。某些最小的操作系统甚至不提供时间。然而如果想和 C 标准一致的话，所有的 C 实现就必须在合理地显示时间方面下功夫。

推诿词

为了让几乎所有人都能摆脱困境，C 标准包含了足够的推诿词。要和 C 标准一致，一个系统只需要提供和当前的时间和日期，或者和处理器耗费的时间最接近的值。一个厂商可以说 1980 年 1 月 1 日总是任何时间和日期的最佳近似值，顾客可能会对这样低质量的近似值有些争论，但是他们不会争论这是否满足了 C 标准。

这些说明的真正含义是一个程序绝不能太认真地对待时间。它可以询问当前的时间（通过调用 `time`），也可以显示以各种漂亮的格式得到的时间，但是它不能确切地知道时间和日期有很重要的意义。如果遇到一个严格依靠精确的时间标志的程序，就要去仔细地检查标准 C 的每一个实现。

15.2 C 标准的内容

<time.h>

7.12 日期和时间 <time.h>

7.12.1 时间的构成

头文件 <time.h> 定义了两个宏，声明了四种类型和几个操作时间的函数。很多函数都处理表示当前日期（根据公历）和时间的日历时间。有些函数处理本地时间（就是表示某个特定时区的日历时间）和夏令时（是对确定本地时间的算法的临时替代）。本地时区和夏令时都是实现定义的。

NULL
CLOCKS_PER_SEC

定义的宏有 NULL（7.1.6 中有描述）和

    CLOCKS_PER_SEC

它是函数 clock 的返回值的每秒的时间单位数。

size_t
clock_t
time_t

声明的类型有 size_t（7.1.6 中有描述）、

    clock_t

和

    time_t

它们是可以表示时间的算术类型，

struct tm

    struct tm

它保存了一个日历时间的各组成部分，日历时间被称为*细分时间*。这个结构至少应该按照某种顺序包含以下成员。注释部分说明了这些成员的语义和它们一般的取值范围 [137]。

```
int tm_sec;    /* seconds after the minute - [0, 61] */
int tm_min;    /* minutes after the hour - [0, 59] */
int tm_hour;   /* hours since midnight - [0, 23] */
int tm_mday;   /* day of the month - [1, 31] */
int tm_mon;    /* months since January - [0, 11] */
int tm_year:   /* years since 1900 */
int tm_wday;   /* days since Sunday - [0, 6] */
int tm_yday;   /* days since January 1 - [0, 365] */
int tm_isdst;  /* Daylight Saving Time flag */
```

如果夏令时有效，则 tm_isdst 的值为正，如果无效则为零，若信息不可用则为负。

7.12.2 时间操纵函数

clock

7.12.2.1 函数 clock

概述

```
#include <time.h>
clock_t clock(void);
```

说明

函数 clock 确定处理器使用的时间。

返回值

函数 clock 返回程序使用的处理器时间的最佳近似值，这个时间从只和程序启动有关的实现定义的某个时期开始计算。如果要以秒为单位计算，则只需把函数 clock 的返回值除以宏 CLOCK_PER_SEC 即可。如果不能获得使用的处理器时间或者这个值不能表示，则函数返回值 (clock_t)-1[138]。

difftime

7.12.2.2 函数 difftime

概述

```
#include <time.h>
double difftime(time_t time1, time_t time0);
```

说明

函数 difftime 计算两个日历时间之差：time1-time0。

返回值

函数 difftime 返回以秒为单位的 double 类型的差值。

mktime

7.12.2.3 函数 mktime

概述

```
#include <time.h>
time_t mktime(struct tm *timeptr);
```

说明

函数 mktime 把 timeptr 指向的结构中的表示为本地时间的细分时间转换为一个日历时间值，这个值和 time 函数的返回值的编码相同。结构成员 tm_wday 和 tm_yday 的初始值被忽略不管，其他成员的初始值不受上面说明的取值范围的限制[139]。若成功完成，则结构的成员 tm_wday 和 tm_yday 被正确设置，同时其他的成员也被设置来表示指定的日历时间，但是它们的值必须在上面说明的范围内，直到确定了 tm_mon 和 tm_year 后才设置 tm_mday 的最终值。

返回值

函数 mktime 返回表示为 time_t 类型的编码值的日历时间。如果日历时间不能被表示，则函数返回值 (time_t)-1。

例子

计算 2001 年 7 月 4 日是星期几。

```
#include <stdio.h>
#include <time.h>
static const char *const wday[] = {
      "Sunday", "Monday", "Tuesday", "Wednesday",
      "Thursday", "Friday", "Saturday", "-unknown-"
};
struct tm time_str;
/*...*/

time_str.tm_year  = 2001 - 1900;
time_str.tm_mon   = 7 - 1;
time_str.tm_mday  = 4;
time_str.tm_hour  = 0;
time_str.tm_min   = 0;
time_str.tm_sec   = 1;
time_str.tm_isdst = -1;
if (mktime(&time_str) == -1)
      time_str.tm_wday = 7;
printf("%s\n", wday[time_str.tm_wday]);
```

time

7.12.2.4 函数 time

概述

```
#include <time.h>
time_t time(time_t *timer);
```

说明

函数 time 确定当前的日历时间，返回值的编码方式未指定。

返回值

函数 time 返回和当前日历时间最近似的值。如果不能获得日历时间，则返回 (time_t)-1。如果 timer 不是空指针，则返回值赋值给它指定的对象。

7.12.3 时间转换函数

除了函数 strftime 之外，这些函数的返回值都存储在以下某种静态对象中：一个细分时间结构和一个 char 类型的数组。任意函数的执行都可能覆盖其他函数返回的某个静态对象中的信息。实现的行为不受这些函数调用的影响。

asctime

7.12.3.1 函数 asctime

概述

```
#include <time.h>
char *asctime(const struct tm *timeptr);
```

说明

函数 asctime 把 timeptr 指向的结构表示的细分时间转换为以下形式的串。

```
Sun Sep 16 01:03:52 1973\n\0
```

其中使用了与下面等价的算法：

```
char *asctime(const struct tm *timeptr)
{
     static const char wday_name[7][3] = {
          "Sun", "Mon", "Tue", "Wed", "Thu", "Fri" , "Sat"
     };
     static const char mon_name[12][3] = {
          "Jan", "Feb", "Mar", "Apr", "May", "Jun",
          "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
     };
     static char result [26];

     sprintf(result. "%.3s %.3s%3d %.2d:%.2d:%.2d %d\n",
          wday_name[timeptr->tm_wday],
          mon_name[timeptr->tm_mon],
          timeptr->tm_mday, timeptr->tm_hour,
          timeptr->tm_min, timewptr->tm_sec,
          1900 + timeptr->tm_year);
     return result;
}
```

返回值

函数 asctime 返回一个指向串的指针。

ctime

7.12.3.2 函数 ctime

概述

```
#include <time. h>
char *ctime(const time_t *timer);
```

说明

函数 ctime 把 timer 指向的日历时间转换为串形式的本地时间，它等价于：

```
asctime (localtime (timer))
```

返回值

函数 ctime 返回 asctime 把那个细分时间作为参数时返回的指针。

参见： 函数 localtime (7.12.3.4)。

gmtime

7.12.3.3 函数 gmtime

概述

```
#include <time.h>
struct tm *gmtime(const time_t *timer);
```

说明

函数 gmtime 把 timer 指向的日历时间转换为以协调世界时（Coordinated Universal Time，UTC）表示的细分时间。

返回值

函数 gmtime 返回指向那种对象的指针，若 UTC 不可用，则返回一个空指针。

localtime

7.12.3.4 函数 localtime

概述

```
#include <time.h>
struct tm *localtime(const time_t *timer);
```

说明

函数 localtime 把 timer 指向的日历时间转换为表示本地时间的细分时间。

返回值

函数 localtime 返回指向那个对象的指针。

strftime

7.12.3.5 函数 strftime

概述

```
#include <time.h>
size_t strftime(char *s, size_t maxsize,
      const char *format, const struct tm *timeptr);
```

说明

函数 strftime 在 format 指向的串的控制下把字符放入 s 指向的数组中。格式应该是一个多字节字符序列，以它初始的转移状态开始和结束。format 串由零个或者多个转换说明符和普通多字节字符组成。转换说明符由一个 % 字符后面跟一个字符组成，后面的这个字符确定了转换说明符的行为。所有的普通多字节字符（包括终止的空字符）都丝毫不变地复制到数组中。如果复制发生在两个重叠的对象中，则行为未定义。最多可以有 maxsize 个字符放置到数组中。每一个转换说明符都由下面描述的适当字符代替。这个适当的字符由当前区域设置的 LC_TIME 类别和 timeptr 指向的结构中包含的值确定。

"%a" 被区域设置的缩写星期名取代。

"%A" 被区域设置的完整星期名取代。

"%b" 被区域设置的缩写月份名取代。

"%B" 被区域设置的完整月份名取代。

"%c" 被区域设置的适当的日期和时间表示取代。

"%d" 被表示为一个十进制数（01–31）的当月的第几天取代。

"%H" 被表示为一个十进制数（00–23）的小时时间（24 小时制）取代。

"%I" 被表示为一个十进制数（01–12）的小时时间（12 小时制）取代。

"%j" 被表示为一个十进制数（001–366）的当年的第几天取代。

"%m" 被表示为一个十进制数（01–12）的月份取代。

"%M" 被表示为一个十进制数（00–59）的分钟数取代。

"%p" 被区域设置的、与 12 小时制相关的 AM/PM 指示符等价的东西取代。

"%S" 被表示为一个十进制数（00–61）的秒数取代。

"%U" 被表示为一个十进制数（00–53）的当年的第几周（第一个星期日是第一个星期的第一天）取代。

"%w" 被表示为一个十进制数（0–6）的星期几取代，星期日表示为 0。

"%W" 被表示为一个十进制数（00–53）的当年的第几周取代（第一个星期一是第一个星期的第一天）。

"%x" 被区域设置的适当的日期表示取代。

"%X" 被区域设置的适当的时间表示取代。

"%y" 被表示为一个十进制数（00–99）的不带世纪的年份取代。

"%Y" 被表示为一个十进制数的带世纪的年份取代。

"%Z" 被时区名字或者它的缩写取代，如果不能确定时区，则不能被字符取代。

"%%" 被 % 取代。

如果一个转换说明符表不是上面中的某一个，则行为未定义。

返回值

如果包含终止的空字符在内的结果字符的总数不大于 maxsize，则函数 strftime 返回字符数，这些字符被放到 s 指向的数组中但不包含终止的空字符。否则，函数返回零，且数组的内容不确定。

脚注

137. tm_sec 的范围 [0, 61] 允许两个闰秒。

138. 要计算一个程序花费的时间，应该在程序的开始调用 clock 函数，然后用后面的调用的返回值减去它的返回值。

139. 因此，tm_isdst 的正值或者零会使函数 mktime 初始时认为夏令时对指定的时间分别有效或者无效。一个负值会导致函数试图确定夏令时对指定的时间是否有效。

15.3 <time.h> 的使用

<time.h> 中声明的函数可以确定用的处理器时间和日历时间，它们也可以在不同的数据表示间进行转换，可以用以下方式表示时间。

- 类型 clock_t，表示程序使用的处理器时间，作为函数 clock 的返回类型。
- 类型 time_t，表示日历时间，作为函数 time 或者函数 mktime 的返回类型。
- 类型 *double*，用秒表示日历时间，作为函数 difftime 的返回类型。
- 类型 struct tm，表示分成几个独立的组成部分的日历时间，作为函数 gmtime 和 localtime 的返回类型。
- 一个文本串，表示日历时间，作为函数 asctime、ctime 和 strftime 的返回类型。

这里的选择很多，最困难的部分是对一个特殊的应用程序找出想使用的数据表示和函数。

函数 strftime

<time.h> 中声明的一个复杂的函数（至少从外部来看很复杂）是 strftime。它用来在一个格式串的控制下，从 struct tm 中产生一个时间和日期的文本表示。从这种意义上说，它模仿了 <stdio.h> 中的打印函数。它在两个重要的方面和打印函数不同。

- strftime 不接受可变参数表。它从一个参数中就可以获得所有的时间和日期信息。
- strftime 的行为在不同的区域设置之间的变化很大。例如，区域设置类别 LC_TIME 能指定所有日期的文本形式都遵循法国文化的习惯。

例如代码段

```
char buf[100];
strftime(buf, sizeof buf,"%A, %x", localtime(&t0));
```

可能把以下任何一条信息存储在 buf 中：

```
Sunday, 02 Dec 1979
dimanche, le 2 décembre 1979
Weekday 0, 02/12/79
```

如果想按照本地习惯显示时间和日期，那么 strftime 恰好提供了这种灵活性。甚至可以在转换说明符之间编写多字节字符序列。这就意味着允许把日期转换为日文中的汉字和其他的大字符集。

转换说明符

下面是 strftime 定义的转换说明符，每个说明符后面都跟了一个它产生的文本的例子。

- %a 缩写星期名（Sun）。
- %A 完整星期名（Sunday）。
- %b 缩写月份名（Dec）。
- %B 完整月份名（December）。
- %c 日期和时间（Dec 2 06 : 55: 15 1979）。
- %d 当月第几天（02）。
- %H 24 小时制的一天的小时时间（06）。
- %I 12 小时制的一天的小时时间（06）。
- %j 当年第几天，从 001 开始（335）。
- %m 当年的月份，从 01 开始（12）。
- %M 小时后的分钟数（55）。
- %p AM/PM 指示符（AM）。
- %S 分钟后的秒数（15）。
- %U 一年的第几周，从 00 开始，以星期日作为每一周的第一天，（48）。
- %w 星期几，0 为星期日（0）。
- %W 一年的第几周，从 00 开始以星期一作为每一周的第一天，（47）。
- %x 日期（Dec 2 1979）。
- %X 时间（06:55:15）。
- %y 本世纪的年份，年的后两位，00 开始（79）。
- %Y 年份（1979）。
- %Z 时区名，如果有的话（EST）。
- %% 百分号字符（%）

我对 <time.h> 中定义的个别的类型和宏的常见描述进行了总结，后面还对如何使用 <time.h> 中声明的函数进行了简短说明。

共享数据对象　注意这些函数共享两个静态数据对象。所有返回 *char* 型指针的函数都返回指向其中一个数据对象的指针。所有返回 struct tm 型指针的函数都返回指向另一个数据对象的指针。因此，对 <time.h> 中声明的一个函数的调用可能会改变早期对另一个（或者相同的）函数的调用存储的值。如果需要在冲突的函数调用间使用这些值，那么要小心地复制存储在某个共享数据对象中的值。

NULL　NULL 一参考 11.3 节。

CLOCKS_PER_SEC　CLOCKS_PER_SEC——表达式 clock()/CLOCKS_PER_SEC 以秒为单位计算了占用的处理器时间。这个宏可以是任何算术类型，整型或浮点型。如果要保证既可以表示大范围的值又可以表示秒的小数部分，就要把它的类型强制转换为 *double* 类型。

clock_t　clock_t——这是 clock 返回的算术类型，clock 将在下面说明。该类型表示占用的处理器时间，它可以为整型或者浮点型，但没有必要和上面讲的宏 CLOCKS_PER_SEC 的类型相同。

size_t　size_t——参考 11.3 节。

time_t　time_t——这是函数 time 返回的算术类型，time 将在下面说明。有几个 <time.h> 中声明的其他函数也对这个类型的值进行操作。它表示跨越很多年，很可能到最接近的一秒钟（虽然不一定）的日历时间。不能对这种类型的值进行算术运算。

tm　tm——struct tm 类型的结构，表示一个“细分时间”。<time.h> 中声明的一些函数对这个类型的值进行操作。可以访问 struct tm 的某些成员。它的定义类似下面的形式：

```
struct tm {
    int tm_sec;         seconds after the minute (from 0)
    int tm_min;         minutes after the hour (from 0)
    int tm_hour;        hour of the day (from 0)
    int tm_mday;        day of the month (from 1)
    int tm_mon;         month of the year (from 0)
    int tm_year;        years since 1900 (from 0)
    int tm_wday;        days since Sunday (from 0)
    int tm_yday;        day of the year (from 0)
    int tm_isdst;       DST flag
```

这些成员可能会按照不同的顺序排列，也可能包含其他的成员。如果夏令时（DST）有效，那么 DST 标记大于零，若无效则 DST 标记等于零，若不确定则 DST 标记小于零。不确定的状态会促使函数读取这种结构来使它们自己确定 DST 是否有效。

asctime　asctime——（asc 来自于 ASCII，现在看来有点用词不当。）使用这个函数来生成参数（指向一个细分时间）表示的日期的文本形式。这个函数返回一个指向以空字符结尾的字符串的指针，这种串具有 "sun Dec 2 06 :

55 : 15 1979\n" 这样的形式。这和在 "C" 区域设置中用格式串 "%a%c\n" 调用 strftime 等价。如果想得到英语习惯的形式而不管当前的区域设置，则调用 asctime。如果希望得到随区域设置改变的形式，那么就调用 strftime。参考上面讲的关于共享数据对象的警告。

clock clock——这个函数计算占用的处理器时间而不是日历时间。如果无法得到占用的处理器时间，则返回 -1。否则，每一次调用都应该返回一个等于或者大于上一次调用的值，前提是上一次调用也在这个程序的执行过程中。这是计算程序花费时间的最好方法。参考上面描述的宏 CLOCKS_PER_SEC。

ctime ctime——ctime(pt) 等价于表达式 asctime(localtime(pt))。使用这个函数可以把一个日历时间直接转换为不受当前区域设置限制的文本形式。参考上面讲的关于共享数据对象的警告。

difftime difftime——调用 difftime(t1,t0) 是计算两个时间 t1 和 t0 的差值的唯一安全的方法。如果 t1 比 t0 晚，那么结果（以秒为单位）是一个正值。

gmtime gmtime——（gm 来源于 GMT，现在有点不太准确了。）使用这个函数把一个日历时间转换为分解的 UTC 时间。成员 tm_isdst 应当为零。如果想得到本地时间，使用下面的函数 localtime。参考上面关于共享数据对象的警告。

localtime localtime——使用这个函数把一个日历时间转换为分解的本地时间。成员 tm_isdst 应该能反映系统关于这个时间和日期的夏令时所能提供的一切信息。如果想获得 UTC 时间，使用上面的函数 gmtime。参考上面关于共享数据对象的警告。

mktime mktime——这个函数首先把它的参数（一个细分时间）转化为规范的形式。例如，它可以把秒数添加到细分时间的成员 tm_sec 中。这个函数一次从 tm_sec 中减去 60，并使 tm_min 增 1，直到 tm_sec 在 [0, 59] 的范围内。然后这个函数用相似的方式校正 tm_min，接着是每一个较粗糙的时间划分直到 tm_year。它从其他的部分确定 tm_wday 和 tm_yday。很明显，也可以使用分钟、小时、天、月或者年来改变一个细分时间。

然后 mktime 把这个细分时间转换为一个等价的日历时间。它假设这个细分时间表示一个本地时间。如果成员 tm_isdst 比零小，函数就要尽量确定夏令时是否对这个特定的时间和日期有效。否则，它就保留这个标志的原始状态。因此，修改一个日历时间的唯一可靠的方法就是通过调用 localtime 把它转换为一个细分时间，再修改一下适当的成员，然后通过调用 mktime 把结果转换回一个日历时间。

strftime strftime——这个函数生成一个空字符结尾的文本串，这个串包含了指定的时间和日期信息。编写一个格式串参数来指定字面量文本和转换后的时间和日期的混合体。指定一个细分时间来提供编码的时间和日期信息。当前

区域设置中的类别 LC_TIME 决定了每一次转换的行为。我从本节的“转换说明符”部分开始描述了怎样写格式串。参考上面关于共享数据对象的警告。

time　time——这个函数确定当前的日历时间。如果不能获得当前日历时间，则返回 -1。否则，每一次调用都应该返回和上一次调用相同或者比上一次调用晚的时间，前提是这些调用是在同一个程序的执行过程中。该函数是获得当前时间和日期的最好方式。

15.4 <time.h> 的实现

<time.h> 中声明的函数多种多样。很多函数都要努力处理计算和表示时间和日期时涉及的很多奇异的无规则的情况。要了解各种编码技术。

头文件 <time.h>　图 15-1 显示了文件 time.h。和往常一样，它从内部头文件 <yvals.h> 中继承了一些标准头文件中重复使用的定义。第 11 章讨论了宏 NULL 和类型 size_t 定义的实现。

<yvals.h> 也定义了两个描述基本函数 clock 和 time 的宏：

_CPS
- 宏 _CPS 指定了宏 CLOCKS_PER_SEC 的值；

_TBIAS
- 宏 _TBIAS 给出了 time 的返回值和从 1900 年 1 月 1 日起计算得到的值之间的差值，该值以秒为单位。（<time.h> 中没有这个宏名。）

图 15-1 time.h

```
/* time.h standard header */
#ifndef _TIME
#define _TIME
#ifndef _YVALS
#include <yvals.h>
#endif
        /* macros */
#define NULL   _NULL
#define CLOCKS_PER_SEC _CPS
        /* type definitions */
#ifndef _SIZET
#define _SIZET
typedef _Sizet size_t;
#endif
typedef unsigned int clock_t;
typedef unsigned long time_t;
struct tm {
    int tm_sec;
    int tm_min;
    int tm_hour;
    int tm_mday;
    int tm_mon;
    int tm_year;
    int tm_wday;
    int tm_yday;
```

图 15-1
（续）

```
    int tm_isdst;
    };
        /* declarations */
char *asctime (const struct tm *);
clock_t clock (void);
char *ctime(const time_t *);
double difftime (time_t, time_t);
struct tm *gmtime(const time_t *);
struct tm *localtime(const time_t *);
time_t mktime(struct tm *);
size_t strftime(char *, size_t, const char *,
    const struct tm *);
time_t time(time_t *);
#endif
```

这些宏的值很大程度上依赖 clock 和 time 的实现方式。这个实现用 *unsigned int*（clock_t 类型）来表示占用的处理器时间。同时，它用 *unsigned long*（time_t 类型）表示日历时间，这个日历时间从 1900 年 1 月 1 日起对 UTC 秒数进行计数，可以表示从 1900 至少到 2036 年的日期。不管系统提供的是什么，都必须对它进行调整使它和这些习惯相匹配。

宏 _TBIAS 是一个权宜之计，一般情况下设为零。用户提供的 time 版本应该用一个合适的起点来传递日历时间。然而，UNIX 以秒为单位从 1970 年 1 月 1 日开始计算时间。很多 C 的实现都提供遵循这种习惯的 time 函数。如果直接使用这样的 time 函数很方便，那么 <yvals.h> 应该包含下面的定义：

```
#define _TBIAS  ((70 * 365LU + 17 ) * 86400
```

这个表达式计算了 70 年，包含 17 个闰年表示两个起点之间的时间差。在某些地方，<time.h> 中声明的函数通过加上或者减去 _TBIAS 来调整 time_t 类型的值。

函数 time

图 15-2 显示了文件 time.c，它定义了 UNIX 系统的函数 time。和往常一样，我假设存在一个使用保留名字的 C 可调用函数，这个函数执行 UNIX 系统服务。对这个版本的 time 函数，头文件 <yvals.h> 可以把宏 _TBIAS 定义为零。

函数 clock

UNIX 也为函数 clock 提供了一个精确的替代函数。很多 C 实现也模仿 UNIX 这样做。因此，用户可能不需要做额外的工作，只要适当地定义宏 _CPS 就行了。例如，对于一个 PC 兼容机，这个值大约是 18.2。

图 15-3 显示了文件 clock.c，它定义了 clock 的一个版本，如果操作系统不提供独立地计算占用的处理器时间的服务，可以使用它。这个函数只是返回一个截短版本的日历时间。这种情况下，头文件 <yvals.h> 把宏 _CPS 定义为 1。

图 15-2
time.c

```
/* time function -- UNIX version */
#include <time.h>

        /* UNIX system call */
time_t _Time(time_t *);

time_t (time)(time_t *tod)
    {                                   /* return calendar time */
    time_t t = _Time(NULL) + (70*365LU+17)*86400;

    if (tod)
        *tod = t;
    return (t);
    }
```

图 15-3
clock.c

```
/* clock function -- simple version */
#include <time.h>

clock_t (clock)(void)
    {                                        /* return CPU time */
    return ((clock_t)time(NULL));
    }
```

图 15-4
difftime.c

```
/* difftime function */
#include <time.h>

double (difftime)(time_t t1, time_t t0)
    {                               /* compute difference in times */
    t0 -= _TBIAS, t1 -= _TBIAS;
    return (t0 <= t1 ? (double)(t1 - t0) : -(double)(t0 - t1));
    }
```

函数 difftime

图 15-4 显示了文件 difftime.c。在比较两个时间之前，它要小心地纠正两个时间的偏差。它也要小心地生成两个无符号整数值之间的有符号的差值。注意函数是怎样把差值 t1-t0 在转换为 *double* 类型之后就取反的。

头文件 "xtime.h"

剩下的函数都包含内部头文件 "xtime.h"。图 15-5 显示了文件 xtime.h，它包含了标准头文件 <time.h> 和内部头文件 "xtinfo.h"。（参考 6.4 节的文件 "xtinfo.h"。）那个内部头文件定义了 _Tinfo 类型，也声明了 asctime.c 中定义的数据对象 _Times。（参考 15.4 节。）_Times 指定了 LC_TIME 类别的特定于区域设置的信息。

头文件 "xtime.h" 定义了宏 WDAY，这个宏为 1900 年 1 月 1 日指定了周日以外的某一天（星期一）。该文件定义了 Dstrule 类型，这种类型指定了确定夏令时编码规则的各个组成部分。（参考图 15-9 开始的文件 xgetdst.c。）它也声明了实现这个版本的 <time.h> 的各种内部头文件。

图 15-5
xtime.h

```
/* xtime. h internal header */
#include <time.h>
#include "xtinfo.h"
        /* macros */
#define WDAY    1                       /* to get day of week right * /
        /* type definitions */
typedef struct {
    unsigned char wday, hour, day, mon, year;
    } Dstrule;
        /* internal declarations */
int _Daysto(int, int);
const char *_Gentime(const struct tm *, _Tinfo *,
    const char *, int *, char *);
Dstrule *_Getdst(const char *);
const char *_Gettime(const char *, int, int*);
int _Isdst(const struct tm *);
const char *_Getzone (void);
size_t _Strftime(char *, size_t, const char *,
    const struct tm *, _Tinfo *);
struct tm *_Ttotm (struct tm *, time_t, int);
time_t _Tzoff (void);
```

图 15-6
gmtime.c

```
/* gmtime function */
#include "xtime.h"

struct tm *(gmtime)(const time_t *tod)
    {                   /* convert to Greenwich Mean Time (UTC) */
    return (_Ttotm(NULL, *tod, 0));
    }
```

函数 gmtime

图 15-6 显示了文件 gmtime.c，函数 gmtime 是把用秒表示的日历时间（time_t 类型）转换为一个细分时间（struct tm 类型）的两个函数中较简单的一个。它只调用内部函数 _Ttotm。第一个参数是一个空指针，它告诉 _Ttotm 把细分时间存储在公共的静态数据对象中。第三个参数是零，它强调夏令时无效。

函数 _Ttotm

图 15-7 显示了文件 xttotm.c。它定义了函数 _Ttotm，这个函数承担了把秒转换为年、月、日等等这些繁琐的操作。这个文件也定义了 _Daysto，_Ttotm 和其他的函数用它来进行日历计算。

函数 _Daysto

_Daysto 要计算每年 365 天之外的天数。因此，它必须确定指定的年份和 1900 年之间有多少闰年。函数也计算从一年的开始到指定月份的额外天数。为了实现这一点，有时它还要确定当前的年份是不是闰年。函数可以识别 1900 年不是闰年。它不必为 1800 年和它之前的非闰年担心，或者 2100 年及其以后的年份。（即使如此，只是在这两个界限之间的几十年之内也会产生其他的问题。）

图 15-7
xttotm.c

```
/* _Ttotm and _Daysto functions */
#include "xtime.h"

        /* macros */
#define MONTAB(year)    \
    ((year) & 03 || (year) == 0 ? mos : lmos)

        /* static data */
static const short lmos[] = {0, 31, 60, 91, 121, 152,
    182, 213, 244, 274, 305, 335};
static const short mos[] = {0, 31, 59, 90, 120, 151,
    181, 212, 243, 273, 304, 334};

int _Daysto(int year, int mon)
    {                       /* compute extra days to start of month */
    int days;

    if (0 < year)               /* correct for leap year: 1801-2099 */
        days = (year - 1) / 4;
    else if (year <= -4)
        days = 1 + (4 - year) / 4;
    else
        days = 0;
    return (days + MONTAB(year)[mon]);
    }

struct tm *_Ttotm(struct tm *t, time_t secsarg, int isdst)
    {                       /* convert scalar time to time structure */
    int year;
    long days;
    time_t secs;
    static struct tm ts;

    secsarg += _TBIAS;
    if (t == NULL)
        t = &ts;
    t->tm_isdst = isdst;
    for (secs = secsarg; ; secs = secsarg + 3600)
        {                               /* loop to correct for DST */
        days = secs / 86400;
        t->tm_wday = (days + WDAY) % 7;
        {                                       /* determine year */
        long i;

        for (year = days / 365;
            days < (i = _Daysto(year, 0) + 365L * year); )
            --year;                     /* correct guess and recheck */
        days -= i;
        t->tm_year = year;
        t->tm_yday = days;
        }
```

图 15-7
（续）

```
        {                                          /* determine month */
        int mon;
        const short *pm = MONTAB (year);

        for (mon = 12; days < pm[--mon]; )
            ;
        t->tm_mon = mon;
        t->tm_mday = days - pm[mon] + 1;
          }
        secs %= 86400;
        t->tm_hour = secs / 3600;
        secs %= 3600;
        t->tm_min = secs / 60;
        t->tm_sec = secs % 60;
        if (0 <= t->tm_isdst || (t->tm_isdst = _Isdst(t)) <= 0)
            return (t);                        /* loop only if <0 => 1 */
        }
    }
```

_Daysto 要处理 1900 年以前的年份，只是因为函数 mktime 可以产生那个范围的中间日期并且生成一个可以表示的 time_t 类型的值。（例如，可以从 2000 年开始，后退 2 000 个月，再前进 20 亿秒。）无论参数值是多少，这种方法都能小心地避免整数溢出。这个函数也计算多余的天数而不是全部的天数，所以它可以在不必担心结果溢出的前提下覆盖一个更大范围的年份。

_Ttotm 使用 _Daysto 确定和它的时间参数 secsarg 对应的年份。因为 _Daysto 的反函数不容易实现，所以 _Ttotm 就不断地猜测并迭代。在最坏的情况下，它必须后退一年来验证它的猜测。这两个函数都使用文件的最上面定义的宏 MONTAB 来确定一个指定的月份开始之前的天数。这个宏也假设除了 1900 之外，每 4 年是一个闰年。

_Ttotm 的 isdst（第三个）参数遵守对 struct tm 的成员 isdst 的约定：

- 如果 isdst 比零大，则夏令时一定有效。_Ttotm 认为它的调用者已经对时间参数 secsarg 作了任何必要的调整。
- 如果 isdst 是零，则夏令时一定无效。_Ttotm 认为没有必要对时间参数 secsarg 进行调整。
- 如果 isdst 比零小，调用者不知道夏令时是否有效。_Ttotm 应该努力找出结果。如果这个函数确定夏令时有效，它就把时间提前一个小时（3 600 秒）并且重新计算细分时间。

因此，_Ttotm 将最多循环一次。只有当它需要确定是否需要循环的时候它才调用函数 _Isdst。即使这样，它也只在 _Isdst 断定夏令时有效的时候循环。

函数 _Isdst 图 15-8 显示了文件 xisdst.c，函数 _Isdst 确定夏令时的状态。_Times._Isdst 指向一个说明规则的字符串。（参考图 15-16 文件 asctime.c 中 _Times 的定义，参考 6.4 节对规则字符串的描述。）

_Isdst 使用编码形式的规则工作。那些规则在用户第一次调用函数的时候或者区域设置的变动改变了串 _Times._Isdst 的最后的编码版本时不是现成的。如果那个串是空的，_Isdst 就去找附加在时区信息 _Times._Tzone 后面的规则。它必须调用 _Getzone 来获得时区信息。它调用 _Gettime 来确定所有夏令时规则的开头。如果可能的话，函数 _Getdst 然后对当前的规则数组进行编码。

给定一个编码的规则数组，_Isdst 扫描数组来寻找包含相关年份的规则。它调整由任何工作日（除周日以外）的限制规则指定的天，然后把规则时间和正在测试的时间相比较。注意，对一个指定的开始年份，第一个规则开始时不在 DST 中。同样的年份的后继规则也不一定在 DST 中。

函数 _Getdst 图 15-9 显示了文件 xgetdst.c。它定义了函数 _Getdst，这个函数分析 _Times._Isdst 指向的字符串来构建规则数组。一个（非空）的字符串的第一个字符作为一个区域界定符，和其他提供特定区域设置的时间信息的串一起起作用。这个函数首先对这些界定符计数，以使它可以分配数组。接着它再一次传递这个字符串来对每个独立的域进行分析和检查。

_Getdst 调用内部函数 getint 来使用一个规则转换整型子域。因为这些域中不存在大到可以造成溢出的情况，所以没有溢出检查。这里的逻辑和 _Getdst 本身的逻辑都有点冗长，但简单易懂。

函数 localtime 图 15-10 显示了文件 localtim.c。函数 localtime 像 gmtime 一样调用 _Ttotm。然而，这里，localtime 假设它必须把 UTC 时间转换为本地时间。要做到这一点，这个函数必须以秒为单位计算出 UTC 和本地时区的时间差。

函数 _Tzoff 文件 localtim.c 也定义了函数 _Tzoff，这个函数努力确定这个时间差（tzoff，以分钟为单位）。当用户第一次调用函数时或区域设置的变动改变了串 _Times._Isdst 最后的编码版本时，这个时间差不是当前的。如果那个串是空的，则 _Tzoff 调用函数 _Getzone 来从环境变量中确定时间差（如果可能的话）。

无论是怎样获得的，串 _Times._Tzone 的形式都为 :EST:EDT:+0300。（参考 6.4 节。）_Tzoff 调用函数 _Gettime 来确定第三个域（#2，从零开始计数）的起始位置（p）和长度（n）。<stdlib.h> 中声明的函数 strtol 一定要完全分析这个域来把它转换为一个编码的整数。而且，这个数不能完全没有意义。（最大的数比 12*60 大，因为国际日期变更线的两边都存在一些特殊的时区。）

图 15-8
xisdst.c

```
/* _Isdst function */
#include <stdlib.h>
#include "xtime.h"

int _Isdst(const struct tm *t)
    {               /* test whether Daylight Savings Time in effect */
    Dstrule *pr;
    static const char *olddst = NULL;
    static Dstrule *rules = NULL;

    if (olddst != _Times._Isdst)
        {                                   /* find current dst_rules */
        if (_Times._Isdst[0] == '\0')
            {                               /* look beyond time_zone info */
            int n;

            if (_Times._Tzone[0] == '\0')
                _Times._Tzone = _Getzone();
            _Times._Isdst = _Gettime(_Times._Tzone, 3, &n);
            if (_Times._Isdst[0] != '\0')
                -- _Times._Isdst;               /* point to delimiter */
            }
        if ((pr = _Getdst(_Times._Isdst)) == NULL)
            return (-1);
        free (rules);
        rules = pr;
        olddst = _Times._Isdst;
        }
    {                                       /* check time against rules */
    int ans = 0;
    const int d0 = _Daysto(t->tm_year, 0);
    const int hour = t->tm_hour + 24 * t->tm_yday;
    const int wd0 = (365L * t->tm_year + d0 + WDAY) % 7 + 14;

    for (pr = rules; pr->wday != (unsigned char)-1; ++pr)
        if (pr->year <= t->tm_year)
            {                               /* found early enough year */
            int rday = _Daysto(t->tm_year, pr->mon) - d0
                + pr->day;

            if (0 < pr->wday)
                {                           /* shift to specific weekday */
                int wd = (rday + wd0 - pr->wday) % 7;

                rday += wd == 0 ? 0 : 7 - wd;
                if (pr->wday <= 7)
                    rday -= 7;                  /* strictly before */
                }
            if (hour < rday * 24 + pr->hour)
                return (ans);
            ans = pr->year == (pr + 1)->year ? !ans : 0;
            }
    return (ans) ;
    }
    }
```

15

图 15-9
xgetdst.c

```
/* _Getdst function */
#include <ctype.h>
#include <stdlib.h>
#include <string.h>
#include "xtime.h"

static int getint (const char *s, int n)
    {                                       /* accumulate digits */
    int value;

    for (value = 0; 0 <= --n && isdigit(*s); ++s)
        value = value * 10 + *s - '0';
    return (0 <= n ? -1 : value);
    }

Dstrule *_Getdst(const char *s)
    {                                       /* parse DST rules */
    const char delim = *s++;
    Dstrule *pr, *rules;

    if (delim == '\0')
        return (NULL);
    {                                       /* buy space for rules */
    const char *s1, *s2;
    int i;

    for (s1 = s, i = 2; (s2 = strchr(s1, delim)) != NULL; ++i)
            s1 = s2 + 1;
    if ((rules = malloc(sizeof(Dstrule) * i)) == NULL)
        return (NULL);
    }
    {                                       /* parse rules */
    int year = 0;

    for (pr = rules; ; ++pr, ++s)
        {                                   /* parse next rule */
        if (*s == '(')
            {                               /* got a year qualifier */
            year = getint(s + 1, 4) - 1900;
            if (year < 0 || s[5] != ')')
                break;                      /* invalid year */
            s += 6;
            }
        pr->year = year;
        pr->mon = getint(s, 2) - 1, s += 2;
        pr->day = getint(s, 2) - 1, s += 2;
        if (isdigit(*s))
            pr->hour = getint(s, 2), s += 2;
        else
            pr->hour = 0;
        if (12 <= pr->mon || 99 < pr->day || 99 < pr->hour)
            break;                  /* invalid month, day, or hour */
        if (*s != '+' && *s != '-')
            pr->wday = 0;
```

图 15-9（续）

```
        else if (s[1] < '0' || ' 6' < s[1])
            break;                          /* invalid week day */
        else
            {                           /* compute week day field */
            pr->wday = s[1] == '0' ? 7 : s[1] - '0';
            if (*s == '+')               /* '-' : strictly before */
                pr->wday += 7;              /* '+' : on or after */
            s += 2;
            }
        if (*s == '\0')
            {                               /* done, terminate list */
            (pr + 1)->wday = (unsigned char) -1;
            (pr + 1)->year = year;
            return (rules);
            }
        else if (*s != delim)
            break;
        }
    free(rules);
    return (NULL);
     }
    }
```

图 15-10 localtim.c

```
/* localtime function */
#include <stdlib.h>
#include "xtime.h"

time_t _Tzoff (void)
    {                               /* determine local time offset */
    static const char *oldzone = NULL;
    static long tzoff = 0;
    static const long maxtz = 60*13;

    if (oldzone != _Times._Tzone)
        {                           /* determine time zone offset */
        const char *p, *pe;
        int n;

        if (_Times._Tzone [0] ==' \0' )
            _Times._Tzone = _Getzone();
        p = _Gettime(_Times._Tzone, 2, &n);
        tzoff = strtol(p, (char **)&pe, 10);
        if (pe - p != n
            || tzoff <= -maxtz || maxtz <= tzoff)
            tzoff = 0;
        oldzone = _Times._Tzone;
        }
    return (tzoff * 60);
    }

struct tm *(localtime)(const time_t *tod)
    {                       /* convert to local time structure */
    return (_Ttotm(NULL, *tod + _Tzoff(), -1));
    }
```

15

图 15-11
xgettime.c

```
/* _Gettime function */
#include <string.h>
#include "xtime.h"

const char *_Gettime(const char *s, int n, int *len)
    {                               /* get time info from environment */
    const char delim = *s ? *s++ : '\0';
    const char *s1;

    for (; ; --n, s = s1 + 1)
        {                                   /* find end of current field */
        if ((s1 = strchr(s, delim)) == NULL)
            s1 = s + strlen(s);
        if (n <= 0)
            {                                      /* found proper field */
            *len = s1 - s;
            return (s);
            }
        else if (*s1 == '\0')
            {                                      /* not enough fields */
            *len = 1;
            return (s1);
            }
        }
    }
```

函数 _Gettime　图 15-11 显示了文件 xgettime.c。它定义了函数 _Gettime，这个函数对一个串中的域进行定位，该串指定了特定于区域设置的时间信息。参考上面对 _Getdst 的描述，就会知道 _Gettime 怎样解释域界定符了。如果 _Gettime 不能找到要求的域，它就返回一个指向空串的指针。

函数 _Getzone　图 15-12 显示了文件 xgetzone.c。函数 _Getzone 调用 <stdlib.h> 中声明的函数 getenv 来确定环境变量 "TIMEZONE" 的值，这个值应该和上面描述的特定于区域设置的时间串 _Times._Tzone 格式相同。(可能会使用用来决定夏令时的规则)

"TIMEZONE"
"TZ"
如果 "TIMEZONE" 的值不存在，函数 _Getzone 就会去寻找环境变量 "TZ"。这个值应该遵循 UNIX 格式 EST05ETD。内部函数 reformat 使用 "TZ" 的值在静态缓冲区中生成一种更合适的形式。

如果 _Getzone 不能找到这两个环境变量中的任何一个，它就认为本地时区为 UTC。在任何情况下，它都把它的决定存储在内部静态缓冲区 tzone 中。对这个函数的后续调用返回这个记录的值。因此，环境变量最多被查询一次，就是 _Getzone 第一次调用的那一次。

函数 mktime　图 15-13 显示了文件 mktime.c。函数 mktime 从细分时间 struct　tm 中计算一个整数 time_t。它不遗余力地避免计算过程中的溢出。(如果时间不能被正确地表示，这个函数就被迫返回值 -1。)

图 15-12
xgetzone.c

```
/*_Getzone function */
#include <ctype.h>
#include <stdlib.h>
#include <string.h>
#include "xtime.h"

        /* static data */
static const char *defzone = ":UTC:UTC:0";
static char *tzone = NULL;

static char *reformat (const char *s)
    {                                                   /* reformat TZ */
    int i, val;
    static char tzbuf[] = ":EST:EDT:+0300";

    for (i = 4; 1 <= --i; )
        if (isalpha(*s))
            tzbuf[i] = *s++;
        else
            return (NULL);
    tzbuf[9] = *s == '-' || *s == '+' ? *s++ : '+';
    if (!isdigit(*s))
        return (NULL);
    val = *s++ - '0';
    if (isdigit(*s))
        val = 10 *val + *s++ - '0';
    for (val *= 60, i = 14; 10 <= --i; val /= 10)
        tzbuf[i] = val % 10;
    for (i = 8; 5 <= --i; )
        if (isalpha(*s))
                tzbuf[i] = *s++;
        else
            return (NULL);
    return (*s == '\0' ? tzbuf : NULL);
    }

const char *_Getzone (void)
    {                                   /* get time zone information */
    const char *s;

    if (tzone)
        ;
    else if ((s = getenv("TIMEZONE")) != NULL)
        {                                       /* copy desired format */
        if ((tzone = malloc(strlen(s) + 1)) != NULL)
            strcpy (tzone, s);
        }
    else if ((s = getenv("TZ")) != NULL)
        tzone = reformat(s);
    if (tzone == NULL)
        tzone = (char *)defzone;
    return (tzone);
    }
```
□

图 15-13 mktime.c

```
/* mktime function */
#include <limits.h>
#include "xtime.h"

time_t (mktime)(struct tm *t)
    {           /* convert local time structure to scalar time */
    double dsecs;
    int mon, year, ymon;
    time_t secs;

    ymon = t->tm_mon / 12;
    mon = t->tm_mon - ymon * 12;
    if (mon < 0)
        mon += 12, --ymon;
    if (ymon < 0 && t->tm_year < INT_MIN - ymon
        || 0 < ymon && INT_MAX - ymon < t->tm_year)
        return ((time_t)(-1));
    year = t->tm_year + ymon;
    dsecs = 86400.0 * (_Daysto(year, mon) - 1)
        + 31536000.0 * year + 86400.0 * t->tm_mday;
    dsecs += 3600.0 * t->tm_hour + 60.0 * t->tm_min
        + (double)t->tm_sec;
    if (dsecs < 0.0 || (double)(time_t)(-1) <= dsecs)
        return ((time_t)(-1));
    secs = (time_t)dsecs - _TBIAS;
    _Ttotm(t, secs, t->tm_isdst);
    if (0 < t->tm_isdst)
        secs -= 3600;
    return (secs - _Tzoff());
    }
```
□

图 15-14 ctime.c

```
/* ctime function */
#include <time.h>

char * (ctime)(const time_t *tod)
    {                   /* convert calendar time to local text */
    return (asctime(localtime(tod)));
    }
```
□

图 15-15 strftime.c

```
/* strftime function */
#include "xtime.h"

size_t (strftime)(char *s, size_t n, const char *fmt,
    const struct tm *t)
    {                           /* format time to string */
    return (_Strftime(s, n, fmt, t, &_Times));
    }
```
□

图 15-16 asctime.c

```
/* asctime function */
#include "xtime.h"

		/* static data */
static const char ampm[] = {":AM:PM"};
static const char days[] = {
	":Sun:Sunday:Mon:Monday:Tue:Tuesday:Wed:Wednesday"
	":Thu:Thursday:Fri:Friday:Sat:Saturday"};
static const char fmts[] = {
	"|%b %D %H:%M:%S %Y|%b %D %Y|%H:%M:%S"};
static const char isdst[] = {""};
static const char mons[] = {
	":Jan:January:Feb:February:Mar:March"
	":Apr:April:May:May: Jun: June"
	":Jul:July:Aug:August:Sep:September"
	":Oct:October:Nov:November:Dec:December"};
static const char zone[] = {""};		/* adapt by default */
static _Tinfo ctinfo = {ampm, days, fmts, isdst, mons, zone};
_Tinfo _Times = {ampm, days, fmts, isdst, mons, zone};

char *(asctime)(const struct tm *t)
	{		/* format time as "Day Mon dd hh:mm:ss yyyy\n" */
	static char tbuf[] = "Day Mon dd hh:mm:ss yyyy\n";

	_Strftime(tbuf, sizeof (tbuf), "%a %c\n", t, &ctinfo);
	return (tbuf);
	}
```

mktime 的第一部分确定年份和月份。如果它们能作为 *int* 型值表示，这个函数就调用 _Daysto 来校正从 1900 年以来的闰天数，然后以秒为单位累加出 *double* 型的时间，以使整数溢出的可能性降到最小。如果最后的值可以用 time_t 类型表示，函数就把它转换为这种类型。mktime 调用 _Ttotm 以使细分时间规范化。最后，函数为夏令时校正时间，并把它从本地时间转换为 UTC。（最后的代码读起来要比编写它简单得多。）

时间格式化函数

<time.h> 中声明的剩余函数通过各种不同的形式把编码的时间转换为文本串。最后，所有的函数都要依赖内部函数 _Strftime 来完成实际的转换，不同的只是对区域设置的选择。函数 asctime（和扩展的函数 ctime）用一种固定的格式转换时间，不管区域设置类别 LC_TIME 的当前状态是什么，这种格式总是遵循 "C" 区域设置的习惯。另一方面，函数 strftime 允许指定一种指导细分时间转换的格式。它遵循当前区域设置的习惯。因此，_Strftime 的一个参数指定了要使用的特定于区域设置的时间信息（_Tinfo 类型）。

函数 asctime

图 15-16 显示了文件 asctime.c。它定义了函数 asctime，这个函数以相同的方式对一个细分时间进行格式化，而不管当前的区域设置是什么。这个文件也定义了数据对象 _Times，它指定了特定于区域设置的时间信息。文

件也定义了内部数据对象 ctinfo，这个对象复制 "C" 区域设置的时间信息。

函数 ctime

图 15-14 显示了文件 ctime.c。函数 ctime 简单地调用 localtime，然后调用 asctime，来转换它的 time_t 类型的参数。因此，它总是遵循 "C" 区域设置的习惯。

函数 strftime

图 15-15 显示了文件 strftime.c。函数 strftime 使用存储在 _Times 中的特定于区域设置的时间信息来调用 _Strftime。因此它的行为随区域设置而改变。

函数 _Strftime

图 15-17 显示了文件 xstrftim.c。它定义了内部函数 _Strftime，该函数完成格式化时间信息的所有工作。_Strftime 使用文件 xstrftim.c 顶部定义的宏 PUT 来传递字符。这个宏封装了所需要的逻辑来复制字符，并对它们进行计数，且限制传递的数目。

<stdlib.h> 中声明的内部函数 _Mbtowc，使用每次调用时提供的 _Mbstate 类型的状态存储对多字节字符串格式进行解析。这个问题和 12.4 节描述的 _Printf 的相同。

函数 _Gentime

图 15-18 显示了文件 xgentime.c。它定义了函数 _Gentime，这个函数为 _Strftime 完成实际的转换。函数 _Gentime 主要由一个很大的 *switch* 语句组成，该语句独立地处理每一个转换。

每一个转换确定一个指针 p，它指向一个字符序列，这个序列给出了转换的结果。它也存储了一个对 *pn 进行计数的有符号整数。一个正的计数值指示 _Strftime 生成指定的字符序列。

产生字符的一个来源是函数 _Gettime，这个函数从特定于区域设置的时间信息的一个串中选择一个域。另一个来源是内部函数 getval（也在文件 xgentime.c 中定义）它产生十进制整数。getval 将那些字符存储在 _Strftime 提供的累加器中。

注意 _Gentime 包含了一个不规范的加法。转换说明符 %D 在转换月份中的天数时，使用第一位空格来代替第一位的 0，这也是 asctime 强调的。

_Gentime 返回一个负的计数值来指示 _Strftime 把一个特定于区域设置转换的格式串“压入”栈中。有 3 个转换随区域设置而改变——%c、%x 和 %X。（例如，转换符% x 在 "C" 区域设置中就变成了 "%b %d %Y"）可以把这些转换表示为调用其他转换的格式串。（6.4 节描述了如何编写一个可以更改这些格式串的区域设置文件。）注意函数 _Strftime 只支持一层栈帧空间。

文件 xgentime.c 中的另一个内部函数是 wkyr。它计算到一年中一个给定的日期的星期数（从那一年开始）。一周可以从星期日（wstart 是 0）或者星期一（wstart 是 1）开始。这种特殊的逻辑避免了取余（模）或者除运算符的参数为负值。

图 15-17
xstrftim.c

```
/* _Strftime function */
#include <stdlib.h>
#include <string.h>
#include "xtime.h"

        /* macros */
#define PUT(s, na) (void)(nput = (na), \
    0 < nput && (nchar += nput) <= bufsize ? \
        (memcpy(buf, s, nput), buf += nput) : 0)

size_t _Strftime(char *buf, size_t bufsize, const char *fmt,
    const struct tm *t, _Tinfo *tin)
    {                                   /* format time information */
    const char *fmtsav, *s;
    size_t len, lensav, nput;
    size_t nchar = 0;

    for (s = fmt, len = strlen(fmt), fmtsav = NULL; ; fmt = s)
        {                                   /* parse format string */
        int n;
        wchar_t wc;
        _Mbsave state = {0};

        while (0 < (n = _Mbtowc(&wc, s, len, &state)))
            {                               /* scan for '%' or '\0' */
            s += n, len -= n;
            if (wc == '%')
                break;
            }
        if (fmt < s)                        /* copy any literal text */
            PUT(fmt, s - fmt - (0 < n ? 1 : 0));
        if (0 < n)
            {                               /* do the conversion */
            char ac[20];
            int m;
            const char *p = _Gentime(t, tin, s++, &m, ac);

            --len;
            if (0 <= m)
                PUT(p, m);
            else if (fmtsav == NULL)
                fmtsav = s, s = p, lensav = len, len = -m;
            }
        if (0 == len && fmtsav == NULL || n < 0)
            {                   /* format end or bad multibyte char */
            PUT ("", 1);                    /* null termination */
            return (nchar <= bufsize ? nchar - 1 : 0);
            }
        else if (0 == len)
            s = fmtsav, fmtsav = NULL, len = lensav;
        }
    }                                                            □
```

15

图 15-18
xgentime.c

```
/* _Gentime function */
#include "xtime.h"

        /* macros */
#define SUNDAY 0                                /* codes for tm_wday */
#define MONDAY 1

static char *getval(char *s, int val, int n)
    {                                   /* convert a decimal value */
    if (val < 0)
        val = 0;
    for (s += n, *s = '\0'; 0 <= --n; val /= 10)
        *--s = val % 10 + '0';
    return (s);
    }

static int wkyr(int wstart, int wday, int yday)
    {                                         /* find week of year */
    wday = (wday + 7 - wstart) % 7;
    return (yday - wday + 12) / 7 - 1;
    }

const char *_Gentime(const struct tm *t, _Tinfo *tin,
    const char *s, int *pn, char *ac)
    {                                       /* format a time field */
    const char *p;

    switch (*s++)
        {                         /* switch on conversion specifier */
    case 'a':                             /* put short weekday name */
        p = _Gettime(tin->_Days, t>_tm_wday << 1, pn);
        break;
    case 'A':                              /* put full weekday name */
        p = _Gettime(tin->_Days, (t>_tm_wday << 1) + 1, pn);
        break;
    case 'b':                               /* put short month name */
        p = _Gettime(tin->_Months, t->tm_mon << 1, pn);
        break;
    case 'B':                                /* put full month name */
        p = _Gettime(tin->_Months, (t->tm_mon << 1) + 1, pn);
        break;
    case 'c':                                 /* put date and time */
        p = _Gettime(tin->_Formats, 0,  pn), *pn = -*pn;
        break;
    case 'd':                        /* put day of month, from 01 */
        p = getval(ac, t->tm_mday, *pn = 2);
        break;
    case 'D':                         /* put day of month, from 1 */
        p = getval(ac, t->tm_mday, *pn = 2);
        if (ac[0] == '0')
            ac[0] = ' ';
        break;
    case 'H':                          /* put hour of 24-hour day */
        p = getval(ac, t->tm_hour, *pn = 2);
        break;
```

图 15-18
（续）

```
    case 'I':                           /* put hour of 12-hour day */
        p = getval(ac, t->tm_hour % 12, *pn = 2);
        break;
    case 'j':                          /* put day of year, from 001 */
        p = getval(ac, t->tm_yday + 1, *pn = 3);
        break;
    case 'm':                         /* put month of year, from 01 */
        p = getval(ac, t->tm_mon + 1, *pn = 2);
        break;
    case 'M':                         /* put minutes after the hour */
        p = getval(ac, t->tm_min, *pn = 2);
        break;
    case 'p':                                        /* put AM/PM */
        p = _Gettime(tin->_Ampm, 12 <= t->tm_hour, pn);
        break;
    case 'S':                       /* put seconds after the minute */
        p = getval(ac, t->tm_sec, *pn = 2);
        break;
    case 'U':                        /* put Sunday week of the year */
        p = getval(ac,
            wkyr(SUNDAY, t->tm_wday, t->tm_yday),*pn = 2);
        break;
    case 'w':                       /* put day of week, from Sunday */
        p = getval(ac, t->tm_wday, *pn = 1);
        break;
    case 'W':                       /* put Monday week of the year * /
        p = getval (ac,
            wkyr(MONDAY, t->tm_wday, t->tm_yday),*pn = 2);
        break;
    case 'x':                                          /* put date */
        p = _Gettime(tin->_Formats, 1, pn), *pn = -*pn;
        break;
    case 'X':                                          /* put time */
        p = _Gettime(tin->_Formats, 2, pn), *pn = -*pn;
        break;
    case 'y':                            /* put year of the century */
        p = getval(ac, t->tm_year % 100, *pn = 2);
        break;
    case 'Y':                                          /* put year */
        p = getval(ac, t->tm_year + 1900, *pn = 4);
        break;
    case 'Z':                                /* put time zone name */
        if (tin->_Tzone[0] == '\0')
            tin->_Tzone = _Getzone();                 /* adapt zone */
        p = _Gettime(tin->_Tzone, 0 < t->tm_isdst, pn);
        break;
    case '%':                                           /* put "%" */
        p = "%", *pn = 1;
        break;
    default:                            /* unknown field, print it */
        p = s - 1, *pn = 2;
        }
    return (p);
    }
```
□

15.5　<time.h> 的测试

图 15-19 显示了文件 ttime.c。测试程序对 <time.h> 中声明的所有函数执行基本的测试。作为质量检查，它也显示了运行程序时函数 time 的日期和时间的返回值。如果一切顺利，这个程序将有类似以下的输出：

```
Current date -- Sun Dec 2 06:55:15 1979
SUCCESS testing <time.h>
```

图 15-19 ttime.c

```
/* test time functions */
#include <assert.h>
#include <stdio.h>
#include <string.h>
#include <time.h>

int main()
    {                           /* test basic workings of time functions */
    char buf [32];
    clock_t tc = clock();
    struct tm ts1;
    time_t tt1, tt2;
    static char *dstr = "Sun Dec 2 06:55:15 1979\n";

    tt1 = time(&tt2);
    assert(tt1 == tt2);
    ts1.tm_sec = 15;
    ts1.tm_min = 55;
    ts1.tm_hour = 6;
    ts1.tm_mday = 2;
    ts1.tm_mon = 11;
    ts1.tm_year = 79;
    ts1.tm_isdst = -1;
    tt1 = mktime(&ts1);
    assert(ts1.tm_wday == 0);
    assert(ts1.tm_yday == 335);
    ++ts1.tm_sec;
    tt2 = mktime(&ts1);
    assert(difftime(tt1, tt2) < 0.0);
    assert(strcmp(asctime(localtime(&tt1)), dstr) == 0);
    assert (strftime(buf, sizeof (buf), "%S",
        gmtime(&tt2)) == 2);
    assert(strcmp(buf, "16") == 0);
    assert(tc <= clock());
    fputs("Current date -- ", stdout);
    time(&tt1);
    fputs(ctime(&tt1), stdout);
    puts("SUCCESS testing <time.h>");
    return (0);
    }
```

□

15.6 参考文献

W.M. O'Neil, *Time and the Calendars*, (Sydney, N.S.W.: Sydney University Press, 1975). 历法本来就是很特别的东西，这本书介绍了计算日历时间的历史，它告诉你的可能比你想知道的还多。它也解释了天和日期为什么以今天这种方式命名和确定。

15.7 习题

15.1 写一个区域设置文件，这个文件按照法语的习惯表示时间，你需要改变以下几项。

```
am_pm       days
dst_rules   months
time_zone   time_formats
```

测试新的区域设置。（提示：可以从本章和前几章的测试程序入手。）

15.2 确定你生活的地方的夏令时的开始和结束的规则。（如果你生活的地方不使用夏令时，就选择一个你想居住并且使用夏令时的地方。）写一个遵守这个规则的区域设置文件。在过去的 20 年中，这个规则是怎样改变的？你能为 `dst_rules` 用一个区域设置文件说明来简明地表示所有这些变化吗？

15.3 很多天文学家认为宇宙开始于大约 150 亿年前的大爆炸。从那次大爆炸开始已经过去了多少秒？要表示从大爆炸开始经过的秒数需要用多少位？

15.4 闰年一般在 4 的倍数的年份发生，一般不在 100 的倍数的年份发生，但是会在 400 的倍数的年份发生。修改文件 `xttotm.c` 中定义的函数 `_Daysto`，确定 1801 年之前和 2099 年之后的距离最近的闰年。在哪一段时间能够让这个函数正确工作？

15.5 编写函数 `long delta-days(int year,int mon,int delta-man)`，这个函数计算一定范围的月份中的天数。初始的天是年份 `year` 中的月份 `mon` 的第一天。月份的范围是有符号值 `delta_man`。你为什么需要指定初始的年份？

15.6 为你的系统实现基本函数 `clock` 和 `time`。对于这些函数的返回值的精度（和意义）你有什么看法？

15.7 近年来，天文学家已经在新年前一天的午夜中把“闰秒”加到了某些年份中。（这纠正了地球的减速旋转。）找到添加了闰秒那些年份。在 `time` 函数的适当地方纠正闰秒。

15.8 [**难**] 把世界上所有的时区汇集到一个表中。为这些时区设计一种便于记忆的命名方案。添加一个函数，这个函数可以通过这个助记名字确定你的工作时区。怎么处理夏令时？

15.9 [**很难**] 设计一种使用文本串简洁地表示日历时间的符号，人们可以很容易地打出这些字符串。编写函数 `time_t strtotime(const char *)`，这个函数可以分析一个以空字符结尾的日历时间串，并且生成相对应的编码日历时间。你怎样调整这个符号使它能适合当前的区域设置？

附录 A

接　口

这个附录概括了把C标准库实现和一个具体的执行环境连接到一起所要做的工作。它主要面向那些想使用目前为止我给出的实现做点什么的读者，其他人只有在理解涉及的话题后，才会发现一些有趣的东西。如果你只关心C标准或者给用户的建议，那么可以跳过下面的内容。

即使对于可能的实现人员，对本附录的使用差别也会很大。有些只是希望挖掘一些有用的代码，那么这些人只需要找到满足其要求的相关小节；而其他人可能想用它们完全取代一个现有的C库，那么他们要做更多的工作。这里我只是粗略地指出了那些额外的步骤。

假设

我引用了头文件<yvals.h>来概括尽可能多的参数。在那些不能使用这个头文件的地方，我引用头文件"yfuns.h"来修改那些低级原语的名字。我没有保证只改变这些头文件就可以使这个库适应所有可能的环境。代码受到各种假设的限制。在这些假设不能满足的地方，你必须修改代码来适应这些情况。下面是必须满足的假设：

- 所有文件——回顾 0.4 节开始的假设，库的很多部分都假设用户能在库内部定义可写的静态数据对象。参考 2.4 节的相关讨论。

<ctype.h>

- <ctype.h>——文件 xctype.c、xtolower.c 和 xtoupper.c 假设执行字符集是 ASCII。把它们包含的表修改为一个不同的字符集。这些文件也假设一个 *char* 占据 8 位。如果 *char* 比 8 位大，那就可能要重新考虑以这个表为基础的方法。

<errno.h>

- <errno.h>——文件 errno.c 和 errno.h 假设可以把 errno 作为一个可写的静态数据对象来维护。为了获得延缓的错误报告，可能需要在每次访问 errno 的时候都调用一个函数。

<float.h>

- <float.h>——文件 float.h 和 xfloat.c 假设浮点值的格式是 IEEE 754 或者一个和它接近的相关形式。如果格式有很大不同，那么可能要重新考虑以 <yvals.h> 中的参数为基础的实现方法。

<limits.h>

- <limits.h>——文件 limits.h 假设一个 *char* 变量占据 8 位，一个 *int* 变量占据 2 个或 4 个字节。（参考 5.4 节的相关讨论。）

<locale.h>

- <locale.h>——这些代码假设了库的某几个部分内部工作的知识。如果你修改了 <ctype.h>（转换表）、<limits.h>（MB_LEN_MAX）、<stdlib.h>（多字节函数）、<string.h>（整理函数）或者 <time.h>（特定于区域设置的时间信息）中的任何代码，就可以到这里寻找问题。

<math.h>

- <math.h>——这些代码对浮点格式的依赖程度至少和前面的 <float.h> 相同。（参考从 7.1 节开始的讨论。）如果 *double* 保留了多于 56 位的精度或者以 10 为基数，那么就要准备做些大的修改。

<stdarg.h>

- <stdarg.h>——文件 stdarg.h 假设传递给函数的参数存储在向上分配的存储空间中，它们遵循一种可预见的模式。（参考 10.4 节。）如果这些假设中有任何一个不成立，你必须重新考虑这种方法。

<stddef.h>

- <stddef.h>——文件 stddef.h 中的宏 offsetof 假设读者可以实现某些涉及指针和整数的技巧。（参考 11.4 节。）如果这些技巧不适用，你必须重新找到一个替代的可以工作的技巧集。（这样的集合一定存在。）

原语

有 19个函数在很大程度上依赖于执行环境，可以把它们作为连接实现和执行环境的基本原语对待。我基本上不去尝试提供这些函数的参数化版本，因为在这里需要对它们进行重大修改。在很多情况下，一个 C实现中现有的函数可以代替它们。除非你想完全取代一个现有的C库，否则你就可以使用这些代码，而不用自己重新编写。下面是对这些原语的一些总结：

<setjmp.h>

- <setjmp.h>——对每一个实现来说，函数 setjmp 和 longjmp 必须用汇编语言编写。或者可以通过修改文件 yvals.h 中定义的宏 _NSETJMP 来调整文件 setjmp.h。然而，不要奢望使用示例文件 longjmp.c 和 setjmp.c。

<signal.h>

- <signal.h>——一定要修改文件 raise.c 和 signal.c 来控制硬件信号。某些系统对函数 signal 直接提供了代替函数。

<stdio.h>

- <stdio.h>——有 9 个函数和宏使系统的大部分和代码的其他部分独立。这些函数在文件 remove.c、rename.c、tmpnam.c、xfgpos.c、xfopen.c 和 xfspos.c 中，这些宏是文件 yfuns.h 中定义的 _Fclose、_Fread 和 _Fwrite。某些系统为某些函数直接提供了代替函数。然而，一定要仔细检查，这些候选者在名字正确的同时，功能也要对得上。

<stdlib.h>

- <stdlib.h>——有 4 个函数和宏使系统的大部分和代码的其他部分独立。这些函数在文件 getenv.c、system.c 和 xgetmem.c 中。宏是文件 yfuns.h 中定义的 _Exit。在文件 yfun.h 中正确地定义

或者声明了数据对象 _Envp 以后，你可以经常使用这里给出的文件 getenv.c。

<time.h>

- <time.h>——有两个函数使系统的大部分和代码的其他部分独立。这些函数在文件 clock.c 和 time.c 中。就像我在这里做的这样，可以在 time.c 的基础上编写 clock.c，如果执行环境对消耗的处理器时间没有提供一个独立的测量，这样做就会很方便。

头文件 "yfuns.h"

图 A-1显示了文件yfuns.h，它是头文件"yfuns.h"的一个版本，可以在很多 UNIX系统下工作。该文件遵循了我对以前的 UNIX例子使用的命名规则。下面是一个完整的具有外部连接的名字的列表，在 UNIX下这个实现需要定义这些名字。每一个名字后面是常规的 UNIX库名字：

```
_Environ environ _Lseek lseek
_Clock clock _Open open
_Close close _Read read
_Execl execl _Sbrk sbrk
_Exit exit _Signal signal
_Fork fork _Time time
_Getpid getpid _Unlink unlink
_Kill kill _Write write
_Link link
```

我把 _Environ 放在第一位是因为它命名了一个数据对象。(像定义在 <errno.h> 中的宏 errno 一样，必要的时候，它可以是一个返回指针的函数调用。)其他所有名字的函数都提供 UNIX 系统服务。你也可以编写或者修改汇编语言文件来提供这些服务。

图 A-1 yfuns.h

```
/* yfuns.hfunctions header -- UNIX version */
#ifndef _YFUNS
#define_YFUNS
        /* macros */
#define_Envp (*_Environ)
#define_Fclose(str) _Close((str)->_Handle)
#define_Fread(str, buf, cnt) _Read((str)->_Handle, buf, cnt)
#define  _Fwrite(str,buf. cnt) _Write((str)->_Handle, buf, cnt)
        /* interface declarations */
extern const char **-Environ;
int_Close(int);
void_Exit ( int );
int_Read(int,     unsigned       char *, int);
int_Write(int,    const unsigned char *, int);
#endif
```

可以用常规的名字来取代这些保留名字。那是开始使用这个实现的快速的方式，但是那种捷径也会导致一些命名冲突，并且它也违背了 C标准中关于命名空间使用的规则。

在给定了必要的原语的情况下，通过修改内部头文件<yvals.h>来适应剩余的代码。这个头文件定义了下面的宏：

_ADNBND
- _ADNBND——stdarg.h 使用这个宏来使参数指针后移（值一般为 0、1、3 或 7）。

_AUPBND
- _AUPBND——stdarg.h 使用这个宏来使参数指针前移（值一般为 0、1、3 或 7）。

_C2
- _C2——limits.h 使用这个宏来把 2 的补码表示（值 1）和 1 的补码表示或有符号数值（值 0）区分开。

_CPS
- _CPS——time.h 使用这个宏来确定宏 CLOCKS_PER_SEC 的值。

_CSIGN
- _CSIGN——limits.h 使用它来确定 *char* 是可以表示负值（值非零）还是只能表示正值（零）。

_D0
- _D0——多个文件使用它来确定浮点值存储的字节顺序（值为 0 或 3）。

_DBIAS
- _DBIAS——某几个文件使用它来确定 *double* 的特征值和它的有符号指数的差值。

_DLONG
- _DLONG——某几个文件使用它来确定 *long double* 是 IEEE 754 10 字节格式（值非零）还是和 *double* 相同（零）。

_DOFF
- _DOFF——某几个文件使用它来确定一个 *double* 特征值在最高有效字中的偏移位。

_EDOM
- _EDOM—errno.h 使用它来确定宏 EDOM 的值。

_EFPOS
- _EFPOS——errno.h 使用它来确定宏 EFPOS 的值。

_ERANGE
- _ERANGE——errno.h 使用它来确定宏 ERANGE 的值。

_ERRMAX
- _ERRMAX——errno.h 使用它来确定错误编码的范围。

_FBIAS
- _FBIAS——xfloat.h 使用它来确定一个 *float* 特征值和它的有符号指数之间的差值。

_FNAMAX
- _FNAMAX——stdio.h 使用它来确定宏 FILE_NAME_MAX 的值。

_FOFF
- _FOFF——xfloat.h 使用它来确定一个 *float* 特征值在最高有效字中的偏移位。

_FOPMAX
- _FOPMAX——stdio.h 使用它来确定宏 FOPEN_MAX 的值。

_FRND
- _FRND——float.h 使用它来确定宏 FLT_ROUNDS 的值。

_ILONG
- _ILONG——limits.h 使用它来确定 *int* 占据 32 位（值非零）还是 16 位（零）。

_LBIAS
- _LBIAS——某几个文件使用它来确定一个 *long double* 特征值和它的有符号指数之间的差值。

_LOFF
- _LOFF ——某几个文件使用它来确定一个 *long double* 特征值在最高有效字中的偏移位。

_MBMAX
- _MBMAX——limits.h 使用它来确定宏 MB_LEN_MAX 的值。

_MEMBND
- _MEMBND——某几个文件使用它来指定最坏情况的存储空间边界（值一般为 0、1、3 或 7）。

_NSETJMP
- _NSETJMP——setjmp.h 使用它来确定 *int* jmp_buf 数组的大小。

_NULL
- _NULL——某几个文件使用它来确定宏 NULL 的值（值为 0、0L 或 (void *) 0)。

_SIGABRT
- _SIGABRT——signal.h 使用它来确定宏 SIGABRT 的值。

_SIGMAX
- _SIGMAX——signal.h 使用它来确定信号编码的范围。

_TBIAS
- _TBIAS——某几个文件使用它来纠正表示为 time_t 类型的日历时间的起始点。

_TNAMAX
- _TNAMAX——stdio.h 使用它来确定宏 L_tmpnam 的值。

我会给出几个例子，它们也具有这些参数。

DEC VAX VLTRIX

图 A-2显示了文件 yvals.h，这是头文件 <yvals.h>在VAX ULTRIX系统下工作的版本。大部分参数都适用于很多 UNIX的版本。浮点参数描述了VAX和较老的 PDP-11计算机体系结构支持的特有格式。那种格式并不真正支持 Inf和 NaN的编码，但是无论怎样，这个库定义了它们。只要你不对这些特殊编码执行算术操作，它们还会传达有用的信息。

Sun UNIX 下的 GNU C

可以容易地把yvals.h的这个版本修改为可以在 Sun UNIX（使用 Motorola MC680X0微处理器）下的 GUN C编译器中工作。首先，修改描述 IEEE 754格式的浮点型参数：

```
#define    _D0       0
#define    _DBIAS    0x3fe
#define    _DLONG    0
#define    _DOFF     4
#define    _FBIAS    0x7e
#define    _FOFF     7
#define    _FRND     1
#define    _LBIAS    0x3fe
#define    _LOFF     4
```

然后修改存储对齐参数：

```
#define    _AUPBND  3U
#define    _ADNBND  0U
#define    _MEMBND  3U
```

当然，也必须提供一个重新命名的 UNIX系统服务的集合。

完整的库

如果你的目标是对一个具体的编译器，完全取代一个现有的库，你还要关注以下两点。

- 必须提供一个 C *启动头文件*（startup header），这个头文件开始的时候从操作系统获得控制权。这就要求熟知操作系统如何运行程序的知识。C 启动头文件保证了可以正确地设置调用栈，正确地初始化静态存储空间，且 3 个标准流已经打开。它调用 main 函数，然后用 main 返回的状态调用 exit。不同的操作系统在这些方面提供的帮助参差不齐。

图 A-2 yvals.h

```
/*yvals.h values header -- VAX ULTRIX version */
#define_YVALS
        /* errno properties */
#define_EDOM 33
#define_ERANGE 34
#define_EFPOS 35
#define_ERRMAX 36
        /* float properties */
#define_D0 0
#define_DBIAS 0x80
#define_DLONG 0
#define-DOFF 7
#define_FBIAS 0x80
#define_FOFF 7
#define_FRND 1
#define-LBIAS 0x80
#define_LOFF 7
        /* integer properties */
#define_C2 1
#define_CSIGN 1
#define_ILONG 1
#define_MBMAX 8
typedef unsigned short -Wchart;
        /* pointer properties */
#define_NULL (void *) 0
typedef        int -Ptrdifft;
typedef        unsigned int -Sizet;
        /* setjmp properties */
#define_NSETJMP 80
        /* signal properties */
#define_SIGABRT 6
#define_SIGMAX 32
        /* stdio properties */
#define_FNAMAX 64
#define_FOPMAX 16
#define_TNAMAX 16
        /* stdlib properties */
#define_EXFAIL 1
        /* storage alignment properties */
#define_AUPBND 3U
#define_ADNBND 3U
#define_MEMBND7U
        /* time properties */
#define_CPS 1
#define_TBIAS 0
```
□

- 必须提供所有的 C 运行时（runtime）函数，生成的代码可能调用这些函数。这就要求熟知编译器如何生成代码的知识。例如，一个 *switch* 语句，经常使用内联代码调用一个运行时函数，而不是执行所有的比较和分支。不同的编译器对 C 运行时函数的依赖程度差别很大。

除非你必须要解决许可问题，否则你会发现完全取代ULTRIX或者GNU C库没有任何优势可言。

Borland Turbo C++

我也使用Borland Turbo C++编译器运行了这本书中的代码。（我还使用过其中附带的ANSI C编译器。）你可以自由选择要取代Borland库的多少。你甚至可以在一定程度上获得Borland库源代码。下面是yvals.h使用这个编译器的一个版本：

```
/* yvals.h values header -- Turbo C++ version */
#define    _YVALS
        /* errno properties */
#define    _EDOM 33
#define    _ERANGE 34
#define    _EFPOS 35
#define    _ERRMAX 36
        /* float properties */
#define    _D0 3
#define    _DBIAS 0x3fe
#define    _DLONG 1
#define    _DOFF 4
#define    _FBIAS 0x7e
#define    _FOFF 7
#define    _FRND 1
#define    _LBIAS 0x3ffe
#define    _LOFF 15
        /* integer properties */
#define    _C2 1
#define    _CSIGN 1
#define    _ILONG 0
#define    _MBMAX 8
typedef unsigned short -Wchart;
        /* pointer properties */
#define    _NULL (void *) 0
typedef int _Ptrdifft;
typedef unsigned int _Sizet;
        /* setjmp properties */
#define    _NSETJMP 10
        /* signal properties */
#define    _SIGABRT 22
#define    _SIGMAX 32
        /* stdio properties */
#define    _FNAMAX 64
#define    _FOPMAX 16
#define    _TNAMAX 16
        /* stdlib properties */
#define    _EXFAIL 1
        /* storage alignment properties */
#define    _AUPBND 1U
#define    _ADNBND 1U
#define    _MEMBND 1U
        /* time properties */
#define    _CPS 1
#define    _TBIAS ((70 * 365LU + 17) * 86400)
```

Borland提供的 C启动头文件定义了 abort和 errno。如果要取代它们，就必须获得它们的源代码并且修改它。否则，最大的担心就是 MS-DOS表示文本文件的方式。在 _Fread中必须丢弃（某些）回车，而在 _Fwrite中必须在（某些）换行前加入回车。同时也要更正在_Fgpos和_Fspos中作的某些改动。对剩下的原语而言，在 Borland库中你通常会发现很多版本。

其他系统

其他操作系统与UNIX没有太大关系。这就使它们更难满足 C标准要求的方式。通常，最大的问题是输入/输出模式。如果要让流的行为很健壮，那么结构化为记录和块的文件就必须进行精细的处理。在一个具有记录或块结构的文件中很难正确处理文件定位要求。

IBM Systme/370

IBM Systme/370是一个极端的例子。其上的操作系统所遵循的约定比UNIX要早得多。甚至这些操作系统中最简单的一个系统都要求使用特殊的接口来正确地支持标准C。它们中最大的系统可以简单地调用特定于系统的代码，这些代码的大小和复杂度方面都和本书中组合的代码相当。这种情况下你绝对希望构建其他操作系统下的工作方法。

独立式程序

如果你的目标是使用这个库生成独立式程序，那么你还要关注其他的东西。你没有操作系统可以依靠，或者最好是一个不完整的操作系统。同一个计算机体系结构下的现有的 C交叉编译器可能给你提供 C启动代码和一个修改后可适应独立环境的 C运行时。一个设计为只生成托管程序的编译器就需要你做这两个方面的工作。

你必须提供的很多原语通常是一个独立环境的存根。例如，考虑一个只支持通过一个端口进行字符的顺序输入和输出的执行环境。函数 _Fread和_Fwrite只需要处理这个端口，_Fgpos、_Fopen等函数对任何参数都可以失败。如果你的需求是适度的，你可以去掉这里的很多难点。

改进

你可能也希望进行各种改进。例如，可以添加错误编码（到 errno.h和strerror.c中）。你可以添加信号编码（到 signal.h和 raise.c中）。你可以实现各种区域设置，甚至可以把较常用的区域设置直接构建到库中。你可以编写 div和 strlen等函数的增强版。这个清单是无穷无尽的，所以这里就不多说了。但是你可以继续探索。祝你好运！

附录B 名字

这个附录列出了库的这个实现定义的实体名字，这些实体都有外部连接或者在其中一个标准头文件中定义。它们是程序可以看到的名字，不管这对你的程序是好还是坏。出现了两次的函数名说明在声明它的标准头文件中有一个屏蔽它的声明的宏定义。

	名字	头文件	文件	所在页
	BUFSIZ	<stdio.h>	stdio.h	276
	CHAR_BIT	<limits.h>	limits.h	76
	CHAR_MAX	<limits.h>	limits.h	76
	CHAR_MIN	<limits.h>	limits.h	76
	CLOCKS_PER_SEC	<time.h>	time.h	424
D	DBL_DIG	<float.h>	float.h	66
	DBL_EPSILON	<float.h>	float.h	66
	DBL_MANT_DIG	<float.h>	float.h	66
	DBL_MAX	<float.h>	float.h	66
	DBL_MAX_10_EXP	<float.h>	float.h	66
	DBL_MAX_EXP	<float.h>	float.h	66
	DBL_MIN	<float.h>	float.h	66
	DBL_MIN_10_EXP	<float.h>	float.h	66
	DBL_MIN_EXP	<float.h>	float.h	66
	EDOM	<errno.h>	errno.h	53
	EFPOS	<errno.h>	errno.h	53
	EOF	<stdio.h>	stdio.h	276
	ERANGE	<errno.h>	errno.h	53
	EXIT-FAILURE	<stdlib.h>	stdlib.h	354
	EXIT_SUCCESS	<stdlib.h>	stdlib.h	354
F	FILE	<stdio.h>	stdio.h	276
	FILENAME_MAX	<stdio.h>	stdio.h	276
	FLT_DIG	<float.h>	float.h	66
	FLT_EPSILON	<float.h>	float.h	66
	FLT_MANT_DIG	<float.h>	float.h	66
	FLT_MAX	<float.h>	float.h	66
	FLT_MAX_10_EXP	<float.h>	float.h	66
	FLT_MAX_EXP	<float.h>	float.h	66
	FLT_MIN	<float.h>	float.h	66
	FLT_MIN_10_EXP	<float.h>	float.h	66

名字	头文件	文件	所在页
FLT_MIN_EXP	<float.h>	float.h	66
FLT_RADIX	<float.h>	float.h	66
FLT_ROUNDS	<float.h>	float.h	66
FOPEN_MAX	<stdio.h>	stdio.h	276
HUGE_VAL	<math.h>	math.h	138
INT_MAX	<limits.h>	limits.h	76
INT_MIN	<limits.h>	limits.h	76
LC_ALL	<locale.h>	locale.h	96
LC_COLLATE	<locale.h>	locale.h	96
LC_CTYPE	<locale.h>	locale.h	96
LC_MONETARY	<locale.h>	locale.h	96
LC-NUMERIC	<locale.h>	locale.h	96
LC-TIME	<locale.h>	locale.h	96
LDBL_DIG	<float.h>	float.h	66
LDBL_EPSILON	<float.h>	float.h	66
LDBL_MANT_DIG	<float.h>	float.h	66
LDBL_MAX	<float.h>	float.h	66
LDBL_MAX_10_EXP	<float.h>	float.h	66
LDBL_MAX_EXP	<float.h>	float.h	66
LDBL_MIN	<float.h>	float.h	66
LDBL_MIN_10_EXP	<float.h>	float.h	66
LDBL_MIN_EXP	<float.h>	float.h	66
LONG_MAX	<limits.h>	limits.h	76
LONG_MIN	<limits.h>	limits.h	76
L_tmpnam	<stdio.h>	stdio.h	276
M MB_CUR_MAX	<stdlib.h>	stdlib.h	354
MB_LEN_MAX	<limits.h>	limits.h	76
NULL	<locale.h>	locale.h	96
" "	<stddef.h>	stddef.h	223
" "	<stdio.h>	stdio.h	276
" "	<stdlib.h>	stdlib.h	354
" "	<string.h>	string.h	398
" "	<time.h>	time.h	424
RAND_MAX	<stdlib.h>	stdlib.h	354
SCHAR_MAX	<limits.h>	limits.h	76
SCHAR_MIN	<limits.h>	limits.h	76
SEEK-CUR	<stdio.h>	stdio.h	276
SEEK_END	<stdio.h>	stdio.h	276
SEEK_SET	<stdio.h>	stdio.h	276
SHRT_MAX	<limits.h>	limits.h	76
SHRT_MIN	<limits.h>	limits.h	76
SIGABRT	<signal.h>	signal.h	200
SIGFPE	<signal.h>	signal.h	200
SIGILL	<signal.h>	signal.h	200
SIGINT	<signal.h>	signal.h	200
SIGSEGV	<signal.h>	signal.h	200
SIGTERM	<signal.h>	signal.h	200
SIG_DFL	<signal.h>	signal.h	200

	名字	头文件	文件	所在页
	fgets	<stdio.h>	fgets.c	293
	floor	<math.h>	floor.c	141
	fmod	<math.h>	fmod.c	148
	fopen	<stdio.h>	fopen.c	279
	fpos_t	<stdio.h>	stdio.h	276
	fprintf	<stdio.h>	fprintf.c	301
	fputc	<stdio.h>	fputc.c	296
	fputs	<stdio.h>	fputs.c	300
	fread	<stdio.h>	fread.c	292
	free	<stdlib.h>	free.c	376
	freopen	<stdio.h>	freopen.c	280
	frexp	<math.h>	frexp.c	143
	fscanf	<stdio.h>	fscanf.c	318
	fseek	<stdio.h>	fseek.c	289
	" "	<stdio.h>	stdio.h	276
	fsetpos	<stdio.h>	fsetpos.c	290
	" "	<stdio.h>	stdio.h	276
	ftell	<stdio.h>	ftell.c	290
	" "	<stdio.h>	stdio.h	276
	fwrite	<stdio.h>	fwrite.c	299
g	getc	<stdio.h>	getc.c	290
	" "	<stdio.h>	stdio.h	276
	getchar	<stdio.h>	getchar.c	291
	" "	<stdio.h>	stdio.h	276
	getenv	<stdlib.h>	getenv.c	380
	gets	<stdio.h>	gets.c	294
	gmtime	<time.h>	gmtime.c	427
	isalnum	<ctype.h>	ctype.h	37
	" "	<ctype.h>	isalnum.c	37
	isalpha	<ctype.h>	ctype.h	37
	" "	<ctype.h>	isalpha.c	38
	iscntrl	<ctype.h>	ctype.h	37
	" "	<ctype.h>	iscntrl.c	38
	isdigit	<ctype.h>	ctype.h	37
	" "	<ctype.h>	isdigit.c	38
	isgraph	<ctype.h>	ctype.h	37
	" "	<ctype.h>	isgraph.c	38
	islower	<ctype.h>	ctype.h	37
	" "	<ctype.h>	islower.c	38
	isprint	<ctype.h>	ctype.h	37
	" "	<ctype.h>	isprint.c	38
	ispunct	<ctype.h>	ctype.h	37
	" "	<ctype.h>	ispunct.c	39
	isspace	<ctype.h>	ctype.h	37
	" "	<ctype.h>	isspace.c	39
	isupper	<ctype.h>	ctype.h	37
	" "	<ctype.h>	isupper.c	39

	名字	头文件	文件	所在页
	isxdigit	<ctype.h>	ctype.h	37
	" "	<ctype.h>	isxdigit.c	39
	jmp_buf	<setjmp.h>	setjmp.h	187
	labs	<stdlib.h>	labs.c	356
	ldexp	<math.h>	ldexp.c	144
	ldiv	<stdlib.h>	ldiv.c	356
	ldiv_t	<stdlib.h>	stdlib.h	354
	localeconv	<locale.h>	localeco.c	97
	" "	<locale.h>	locale.h	96
	localtime	<time.h>	localtim.c	433
	log	<math.h>	log.c	166
	" "	<math.h>	math.h	138
	log10	<math.h>	log10.c	167
	" "	<math.h>	math.h	138
	longjmp	<setjmp.h>	longjmp.c	189
m	malloc	<stdlib.h>	malloc.c	374
	mblen	<stdlib.h>	mblen.c	366
	" "	<stdlib.h>	stdlib.h	354
	mbstowcs	<stdlib.h>	mbstowcs.c	366
	mbtowc	<stdlib.h>	mbtowc.c	366
	" "	<stdlib.h>	stdlib.h	354
	memchr	<string.h>	memchr.c	399
	memcmp	<string.h>	memcmp.c	399
	memcpy	<string.h>	memcpy.c	400
	memmove	<string.h>	memmove.c	400
	memset	<string.h>	memset.c	400
	mktime	<time.h>	mktime.c	436
	modf	<math.h>	modf.c	143
	offsetof	<stddef.h>	stddef.h	223
	perror	<stdio.h>	perror.c	298
	pow	<math.h>	pow.c	168
	printf	<stdio.h>	printf.c	301
	ptrdiff_t	<stddef.h>	stddef.h	223
	putc	<stdio.h>	putc.c	297
	" "	<stdio.h>	stdio.h	276
	putchar	<stdio.h>	putchar.c	297
	" "	<stdio.h>	stdio.h	276
	puts	<stdio.h>	puts.c	300
q	qsort	<stdlib.h>	qsort.c	356
	raise	<signal.h>	raise.c	202
	rand	<stdlib.h>	rand.c	359
	realloc	<stdlib.h>	realloc.c	377
	remove	<stdio.h>	remove.c	283
	rename	<stdio.h>	rename.c	283
	rewind	<stdio.h>	rewind.c	290
s	scanf	<stdio.h>	scanf.c	319
	setbuf	<stdio.h>	setbuf.c	288

名字	头文件	文件	所在页
setjmp	<setjmp.h>	setjmp.c	188
" "	<setjmp.h>	setjmp.h	187
setlocale	<locale.h>	setlocal.c	102
setvbuf	<stdio.h>	setvbuf.c	289
sig_atomic_t	<signal.h>	signal.h	200
signal	<signal.h>	signal.c	203
sin	<math.h>	math.h	138
" "	<math.h>	sin.c	152
sinh	<math.h>	sinh.c	163
size_t	<stddef.h>	stddef.h	223
" "	<stdio.h>	stdio.h	276
" "	<stdlib.h>	stdlib.h	354
" "	<string.h>	string.h	398
" "	<time.h>	time.h	424
sprintf	<stdio.h>	sprintf.c	302
sqrt	<math.h>	sqrt.c	159
srand	<stdlib.h>	srand.c	359
" "	<stdlib.h>	stdlib.h	354
sscanf	<stdio.h>	sscanf.c	319
stderr	<stdio.h>	stdio.h	276
stdin	<stdio.h>	stdio.h	276
stdout	<stdio.h>	stdio.h	276
strcat	<string.h>	strcat.c	402
strchr	<string.h>	strchr.c	403
strcmp	<string.h>	strcmp.c	402
strcoll	<string.h>	strcoll.c	410
strcpy	<string.h>	strcpy.c	402
strcspn	<string.h>	strcspn.c	403
strerror	<string.h>	strerror.c	406
" "	<string.h>	string.h	398
strftime	<time.h>	strftime.c	436
strlen	<string.h>	strlen.c	403
strncat	<string.h>	strncat.c	401
strncmp	<string.h>	strncmp.c	401
strncpy	<string.h>	strncpy.c	402
strpbrk	<string.h>	strpbrk.c	404
strrchr	<string.h>	strrchr.c	404
strspn	<string.h>	strspn.c	404
strstr	<string.h>	strstr.c	405
strtod	<stdlib.h>	stdlib.h	354
" "	<stdlib.h>	strtod.c	362
strtok	<string.h>	strtok.c	405
strtol	<stdlib.h>	strtol.c	362
strtoul	<stdlib.h>	stdlib.h	354
" "	<stdlib.h>	strtoul.c	361
strxfrm	<string.h>	strxfrm.c	408
system	<stdlib.h>	system.c	380
tan	<math.h>	tan.c	153

名字	头文件	文件	所在页
_Dnorm	"xmath.h"	xdnorm.c	147
_Dscale	"xmath.h"	xdscale.c	146
_Dtento	"xmath.h"	xdtento.c	174
_Dtest	"xmath.h"	xdtest.c	140
_Dunscale	"xmath.h"	xdunscal.c	144
_Dvals	<float.h>	float.h	66
_EDOM	<yvals.h>	yvals.h	450
_EFPOS	<yvals.h>	yvals.h	450
_ERANGE	<yvals.h>	yvals.h	450
_ERRMAX	<yvals.h>	yvals.h	450
_ERRNO	<errno.h>	errno.h	53
_Exp	"xmath.h"	xexp.c	160
_FBIAS	<yvals.h>	yvals.h	450
_FLOAT	<float.h>	float.h	66
_FNAMAX	<yvals.h>	yvals.h	450
_FOFF	<yvals.h>	yvals.h	450
_FOPMAX	<yvals.h>	yvals.h	450
_FRND	<yvals.h>	yvals.h	450
_Fgpos	<stdio.h>	xfgpos.c	285
_Files	<stdio.h>	xfiles.c	279
_Flt	<float.h>	xfloat.c	68
_Fmtval	xfmtval.c	92	
_Fopen	"xstdio.h"	xfopen.c	284
_Foprep	"xstdio.h"	xfoprep.c	281
_Freeloc	"xlocale.h"	xfreeloc.c	118
_Frprep	"xstdio.h"	xfrprep.c	295
_Fspos	<stdio.h>	xfspos.c	286
_Fwprep	"xstdio.h"	xfwprep.c	297
_Genld	"xstdio.h"	xgenld.c	316
_Gentime	"xtime.h"	xgentime.c	440
_Getdst	"xtime.h"	xgetdst.c	432
_Getfld	"xstdio.h"	xgetfld.c	324
_Getfloat	"xstdio.h"	xgetfloa.c	328
_Getint	"xstdio.h"	xgetint.c	326
_Getloc	"xlocale.h"	xgetloc.c	104
_Getmem	"xalloc.h"	xgetmem.c	375
_Gettime	"xtime.h"	xgettime.c	434
_Getzone	"xtime.h"	xgetzone.c	435
_Hugeval	<math.h>	xvalues.c	139
_ILONG	<yvals.h>	yvals.h	450
_IOFBF	<stdio.h>	stdio.h	276
_IOLBF	<stdio.h>	stdio.h	276
_IONBF	<stdio.h>	stdio.h	276
_Inf	"xmath.h"	xvalues.c	139
_Isdst	"xtime.h"	xisdst.c	431
_LBIAS	<yvals.h>	yvals.h	450
_LIMITS	<limits.h>	limits.h	76
_LO	<ctype.h>	ctype.h	37

_G

	名字	头文件	文件	所在页
	_LOCALE	<locale.h>	locale.h	96
	_LOFF	<yvals.h>	yvals.h	450
	_Ldbl	<float.h>	xfloat.c	68
	_Ldtob	"xstdio.h"	xldtob.c	312
	_Ldunscale	"xmath.h"	xldunsca.c	172
	_Litob	"xstdio.h"	xlitob.c	310
	_Loctab	"xlocale.h"	xloctab.c	117
	_Locterm	"xlocale.h"	xlocterm.c	122
	_Locvar	"xlocale.h"	xlocterm.c	122
	_Log	<math.h>	xlog.c	166
_M	_MATH	<math.h>	math.h	138
	_MBMAX	<yvals.h>	yvals.h	450
	_MEMBND	<yvals.h>	yvals.h	450
	_Makeloc	"xlocale.h"	xmakeloc.c	120
	_Mbcurmax	<stdlib.h>	xstate.c	107
	_Mbsave	<stdlib.h>	stdlib.h	354
	_Mbstate	"xstate.h"	xstate.c	107
	_Mbtowc	<stdlib.h>	xmbtowc.c	367
	_Mbxlen	<stdlib.h>	mblen.c	366
	_Mbxtowc	<stdlib.h>	mbtowc.c	366
	_NATS	<stdlib.h>	stdlib.h	354
	_NCAT	<locale.h>	locale.h	96
	_NERR	<errno.h>	errno.h	53
	_NSETJMP	<yvals.h>	yvals.h	450
	_NSIG	<signal.h>	signal.h	200
	_NULL	<yvals.h>	yvals.h	450
	_Nan	"xmath.h"	xvalues.c	139
	_PU	<ctype.h>	ctype.h	37
	_Poly	"xmath.h"	xpoly.c	151
	_Printf	"xstdio.h"	xprintf.c	304
	_Ptrdifft	<yvals.h>	yvals.h	450
	_Putfld	"xstdio.h"	xputfld.c	308
	_Randseed	<stdlib.h>	rand.c	359
	_Readloc	"xlocale.h"	xreadloc.c	115
	_Rteps	"xmath.h"	xvalues.c	139
_S	_SETJMP	<setjmp.h>	setjmp.h	187
	_SIGABRT	<yvals.h>	yvals.h	450
	_SIGMAX	<yvals.h>	yvals.h	450
	_SIGNAL	<signal.h>	signal.h	200
	_SIZET	<stddef.h>	stddef.h	223
	_SIZET	<stdio.h>	stdio.h	276
	_SIZET	<stdlib.h>	stdlib.h	354
	_SIZET	<string.h>	string.h	398
	_SIZET	<time.h>	time.h	424
	_SP	<ctype.h>	ctype.h	37
	_STDARG	<stdarg.h>	stdarg.h	211
	_STDDEF	<stddef.h>	stddef.h	223
	_STDIO	<stdio.h>	stdio.h	276

_T

附录 C 术语

这个附录列出了那些在本书中有特殊含义的术语。如果你怀疑一个术语的含义可以到这里查看。

A

access（**访问**）——获得存储在一个数据对象中的值或者在这个数据对象中存储一个新值。

address constant expression（**地址常量表达式**）——可以用来初始化某个指针类型的静态数据对象的表达式。

allocated storage（**可控的存储空间**）——在程序运行时获得存储空间的数据对象。

alphabetic character（**字母字符**）——一个小写或大写字母。

alphanumeric character（**字母数字字符**）——一个字母字符或数字。

American National Standards Institute（ANSI）——美国国家标准协会，在美国被授权制定计算机相关标准的组织。

argument（**实参**）——在一个函数调用中为某个形参提供初始值的表达式。

argument-level declaration（**参数级声明**）——在函数定义或者函数原型中对某个参数进行的声明。

arithmetic type（**算术类型**）——整型或者浮点类型。

array type（**数组类型**）——由某些预先指定的重复的数据对象元素组成的数据对象类型。

American Standard Code for Information Interchange（ASCII）——美国标准信息交换码，标准字符集 ISO 646 的美国版本。

assembly language（**汇编语言**）——适用于特定计算机体系结构的程序设计语言。

assertion（**断言**）——对正确的程序一定为真的谓词。

assign（**赋值**）——在一个数据对象中存储一个值。

assigning operator（**赋值操作符**）——把一个值存储在一个数据对象中的操作

符，例如=、+=或++。

assignment-compatible types（**赋值兼容类型**）——在一个赋值操作符的两边都有效的两种数据类型。

asynchronous signal（**异步信号**）——和程序执行没有关联的重要事件，例如一个提示键被按下。

atomic（**原子操作**）——使两个线程同步的一个不可分割的操作。

B

base（**基数**）——在某个位置的数字表示中表示某个数字的权的值，例如以8为基数（8进制）或以10为基数（10进制）。

basic C character set（**基本C字符集**）——表示一个C源文件需要的最小的字符编码集合。

beginning-of-file（**文件开始**）——一个文件中第一个字节之前的文件位置。

benign redefinition（**良性重定义**）——定义一个已经存在的宏的宏定义，这两个定义有相同的符号序列，两对符号之间有空白。

bias（**偏差**）——在浮点表示中，加到一个指数上以产生特征值的值。

binary（**二进制**）——和文本相对，可以包含任意样式的位组合。

binary stream（**二进制流**）——可以包含任意的二进制数据的流。

block（**块**）——C函数中用一对大括号括起来的一组语句。

block-level declaration（**块级别声明**）——块内声明。

buffer（**缓冲区**）——用作方便的工作区或临时存储空间的数组数据对象，经常用于一个程序和一个文件之间。

C

C Standard（**C标准**）——ANSI和ISO采用的C程序设计语言描述，用来使C实现和程序之间的差别最小。

call tree（**调用树**）——说明程序内部的一组函数之间相互调用关系的层次表。

calling environment（**调用环境**）——保留在栈帧中代表调用函数的信息。

category（**类别**）——区域设置的一部分，用来处理某组服务，例如字符分类或者时间和日期格式。

character（**字符**）——C中占据一个字节的存储空间的数据类型，可以表示基本C字符集中的所有编码。

character class（**字符类**）——相关字符编码的集合，例如数字、大写字符或标点符号。

character constant（**字符常量**）——C程序中的一个记号，例如'a'，它的整

数值是执行字符集中某个字符的编码。

characteristic（**特征值**）——浮点表示的一部分，用来存储指数的偏移量。

close（**关闭**）——终止流和文件之间的连接。

code（**代码**）——对程序设计语言文本或者从文本产生的可执行文件的通俗叫法。

collate（**整理**）——按照某种规则确定两个串的顺序。

compiler（**编译器**）——生成一个可执行文件的翻译器。

computer architecture（**计算机体系结构**）——可以执行常用的可执行文件格式的一类计算机。

constant type（**常量类型**）——初始化后值不能改变的数据对象类型，因为具有 const 类型修饰符。

control character（**控制字符**）——执行空格或者其他控制功能的字符，不在显示设备上显示图像。

conversion specification（**转换说明**）——在打印或者扫描格式中以一个百分号开始的字符序列，它指定了下一次执行的转换或者传输。

conversion specifier（**转换说明符**）——转换说明中的最后一个字符，确定要执行的转换或者传输的类型。

converting type（**转换类型**）——修改一个某种类型的值的表示方法，使它可以表示为另一种类型的值。

cross compiler（**交叉编译器**）——在一种计算机体系结构下执行，且能生成可以在另一种计算机体系结构下使用的可执行文件的编译器。

currency symbol（**货币符号**）——用来显示以标识一种货币量的字符序列，例如 $。

D data object（**数据对象**）——内存中可以存储指定类型值的一组连续的字节。

data object type（**数据对象类型**）——和函数类型相对，描述数据对象的类型。

Daylight Savings Time（**夏令时**）——日历年中的某段时间，在这段时间内，本地时区相对 UTC（协调世界时）向东移一个小时。

decimal（**十进制**）——以 10 为基数的数字表示。

decimal point（**小数点**）——小数中分隔整数部分和小数部分的字符。

declaration（**声明**）—— C 程序中的一个记号序列，用来命名一个名字、为一个数据对象分配空间、定义一个数据对象的初始值、定义一个函数的行为和 / 或指定一种类型。

default（**默认**）——当要求作出选择而没有指定任何选择时的选择。

definition（**定义**）——为一个数据对象分配空间的声明、指定一个函数行为的声明、命名一种类型的声明或者对宏的 *define* 指示。

device handler（**设备处理器**）——操作系统的一部分，用于控制某个 I/O 设备的操作。

diagnostic（**诊断**）—— C 翻译器发出的报告一个无效程序的消息。

digit（**数字**）——用来表示数的 10 个字符中的一个，例如 3。

domain error（**定义域错误**）——用一个（或多个）实参值（或者值）调用一个数学函数，对这个值来说这个函数没有定义。

dot（**点**）——字符 .，经常用作小数点。

dynamic storage（**动态存储空间**）——在进入一个块（或者函数）时分配存储空间，块的活动终止时又把存储空间释放掉的数据对象，例如函数形参、auto 声明和 register 声明。

E　Extended Binary-Coded Decimal Interchange Code（EBCDIC）——扩展的二进制编码的十进制交换码，IBM 广泛使用的字符编码，特别是在 System/370 下。

element（**元素**）——数组数据对象重复的组成部分中的一个。

end-of-file（**文件结束**）——文件中最后一个字节后面的文件位置。

end-of-file indicator（**文件结束符**）—— FILE 数据对象的一个成员，记录上一次读操作是否遇到了文件结束位置。

environment（**环境**）——操作系统提供的 C 程序之外的那些服务，对 C 程序可见，例如文件和环境变量。

environment variable（**环境变量**）——能通过环境和一个串联系在一起的名字。

error indicator（**错误指示符**）—— FILE 数据对象的一个成员，记录上一次操作是否发生了错误。

exception（**异常**）——在程序执行过程中发生的要求特殊处理的情况，例如浮点下溢。

executable file（**可执行文件**）——不需要进一步翻译或者解释，操作系统就可以直接运行的文件。

execution character set（**执行字符集**）——程序执行时使用的字符集。

exponent（**指数**）——浮点值的一个组成部分，指定了底数的幂。

expression（**表达式**）—— C 程序中连续的记号序列，指定了怎样计算某个值和产生副作用。

F field（**域**）——一组连续的字符，和一个扫描格式转换说明指定的样式匹配。

file（**文件**）——有一个文件名的连续的字节序列，由环境维护。

file descriptor（**文件描述符**）——一个非负整数，指定一个由 C 程序打开的一个文件。

file-level（**文件级别**）—— C 源文件中所有声明之外的部分。

file-position indicator（**文件定位符**）——和一个打开的文件相关联的编码值，指定了文件中下一个要读或者写的字节。

file-positioning error（**文件定位错误**）——一个不能完成的改变文件定位符的请求。

file-positioning function（**文件定位函数**）——那些读取或者修改文件定位符的函数。

file name（**文件名**）—— C 标准库中某些函数用来指示一个文件的名字。

finite-state machine（**有限状态机**）——行为由一个状态值和一些谓词决定的计算，例如一个输入值是否和某些指定的值匹配。

floating-point type（**浮点类型**）—— *float*、*double* 或者 *long double* 中的任意一种。

format（**格式**）——一个以空字符结尾的串，用来确定打印、扫描或者时间函数的行为。

formatted input（**格式化输入**）——在一个格式的控制下读取文本并把它转换为编码值，就像扫描函数那样。

formatted output（**格式化输出**）——在一个格式的控制下转换编码值并把它们作为文本输出，就像打印函数那样。

fraction（**小数部分**）——浮点值的一个组成部分，按照确定的精度描述区间[1/base,1) 之间的一个值。

free（**释放**）——释放前面的程序执行为某个数据对象分配的存储空间。

function（**函数**）——一组连续的可执行语句，当它在一个表达式内调用的时候，接受和它的形参相对应的实参值，并（可能）返回供那个表达式使用的一个值。

function prototype（**函数原型**）——包括对函数形参的强制声明的函数声明。

G Greenwich Mean Time（GMT）——格林尼治平时，UTC 的旧称。

GNU C——马萨诸塞州的一个组织开发的可移植 C 编译器，这个组织的软件应用范围很广。

graphic（**图形**）——一个打印字符的可见表示。

H handle（**句柄**）——文件描述符的另一种叫法。

header file（**头文件**）——一个文本文件，通过在 C 源文件中被 `#include` 预处理指令包含而成为翻译单元的一部分。

heap（**堆**）——内存的一部分，可执行程序可以使用它存储分配的数据对象。

hexadecimal（**十六进制**）——以 16 为基数的数字表示。

hole（**空隙**）——一个数据对象或者参数表内部，对确定它们的值不起作用的一组连续的位或者字节。

I identifier（**标识符**）——一个名字。

Institute of Electrical and Electronic Engineers（IEEE）——美国电气及电子工程师协会，ANSI 授权的开发计算机相关的标准的组织之一。

implementation（**实现**）——一种规范说明的可工作版本，例如一种程序设计语言。

include file（**包含文件**）——一个文本文件，通过在一个 C 源程序或者另一个包含文件中被 `#include` 预处理指令包含而成为翻译单元的一部分。

infinity（**无穷大**）——表示一个值太大了而不能被有穷值表示的一个浮点编码。

integer（**整数**）——不含分数或小数的数，可能为负数或者零。

integer constant expression（**整数常量表达式**）——翻译器翻译的时候可以将其缩减为一个已知的整数值的表达式。

integer type（**整型**）——可以表示某些连续的整数区间（包括零）的数据对象类型。

Intel 80X86——IBM PC 及其兼容机上广泛使用的微处理器家族。

Intel 80X87——支持 Intel 80X86 家族的 IEEE 754 浮点算术标准的数学协处理器家族。

interface（**接口**）——C 程序可用的、提供某种服务（例如输入 / 输出）的函数或者约定的集合。

international currency symbol（**国际货币符号**）——三字母编码加上一个空格或者点，指定世界上某种货币，由 ISO 4217:1987 指定。

interpreter（**解释器**）——程序执行过程中仍保持控制权的翻译器。

invalid（**无效的**）——未遵循 C 标准。

I/O——输入和输出。

ISO——国际标准化组织，负责开发国际计算机相关的标准的组织。

K knock out（**淘汰**）——通过为一个具有外部连接的名字提供一个定义来阻止链接器连接一个库目标模块。

L letter（**字母**）——英语字母表中 a~z 和 A~Z 的 52 个字母之一，加上 "C" 区域设置之外的其他附加字符。

librarian（**库管理器**）——维护目标模块库的程序。

library（**库**）——目标模块的集合，链接器可以有选择地把它和一个可执行程序连接来为具有外部连接的名字提供定义。

linker（**链接器**）——把目标模块组合起来形成可执行文件的程序。

locale（**区域设置**）——修改 C 标准库的行为以适应某个具体的文化或者职业的约定的信息的集合。

locale-specific（**特定于区域设置**）的——因区域设置的变化而变化。

lowercase letter（**小写字母**）——英语字母表中 a~z 这 26 个字母中的一个，还可能会加上 "C" 区域设置之外某些附加的字符。

lvalue（**左值**）——指定一个数据对象的表达式。

M machine（**机器**）——对各种计算机体系结构的通称。

macro（**宏**）——由 #define 预处理指令定义的一个名字，它指定了翻译单元中宏的后续调用的代替文本。

macro definition（**宏定义**）——和一个宏名关联的代替文本。

macro guard（**宏保护**）——用来保证一个文本序列在一个翻译单元中最多被连接一次的宏名。

macro, masking（**宏屏蔽**）——屏蔽一个翻译单元前面声明的相同的名字的宏定义。

member（**成员**）——一个数据对象声明，它指定了一个结构或者联合声明中的其中一个组成部分。

mode（**模式**）——指定了两种或者多种可选行为的修饰符，例如一个打开文件的文本模式和二进制模式。

modifiable lvalue（**可更改的左值**）——指定一个可以在其中存储一个新值的数据对象（既不是常量，也不是数组类型）的表达式。

monetary（**货币的**）——与货币有关的。

Motorola MC680X0——Apple Macintosh 和某些 Sun 工作站下常用的微处理器家族。

Motorola MC68881——支持 Motorola MC680X0 家族的 IEEE 754 浮点算术的数学协处理器家族。

MS-DOS——微软公司为 PC 可兼容的电脑开发的流行的操作系统。

multibyte character（**多字节字符**）——大字符集中的一个字符，通常编码为一个或者多个常规（单字节）字符序列。

multithread（**多线程**）——在一个给定的时间间隔内，支持多个程序执行，可能还允许各个独立程序执行之间的交互。

N

name（**名字**）——在一个翻译单元中用来指定不同的实体——例如函数、宏或者成员——的、一个大集合中的一个记号。

name space（**命名空间**）——在 C 程序内部通过上下文环境可区分的名字集合。

native（**本地**）——以空串 "" 命名的区域设置。

not-a-number（**非数**）——不指定任何数字值的浮点编码，例如一个没有定义的结果。

null character（**空字符**）——编码值为零的字符。

null pointer（**空指针**）——和零比较时相等的指针类型值，因此它不指向任何函数或者数据对象。

null-pointer constant（**空指针常量**）——一个整数常量表达式，例如 0，在某些上下文环境中它可以作为空指针使用。

O

object module（**目标模块**）——翻译单元翻译后的形式，适合作为可执行程序的一部分来连接。

octal（**八进制**）——以 8 为基数的数字表示。

offset（**偏移量**）——数据对象内部的成员或者元素的相对地址，通常以字节为单位。

one's-complement arithmetic（**1 的补码算术**）——一种二进制编码形式，这种形式下，一个数的相反数是它的按位取反。

open（**打开**）——在一个文件和一个流之间建立联系。

operand（**操作数**）——C 表达式中的子表大式，操作符对其进行操作。

operating system（**操作系统**）——一个运行其他程序的程序，通常会掩盖那些具有相同体系结构的计算机之间的差别。

operator（**操作符**）——C 表达式中产生一个给定类型值的记号，并且可能会产生副作用，可以跟一个到 3 个子表达式作为操作数。

overfiow（**上溢**）——一个太大而不能表示为要求的整型或者浮点类型的值的计算。

P parameter（**形参**）——函数中声明的数据对象，它存储函数调用时对应的实参值。

parse（**分析**）——确定一个记号序列的语法结构。

PC——IBM 在 20 世纪 80 年代早期推出的计算机体系结构，它已经变成了使用最广泛的个人电脑。

PDP-11——DEC 计算机体系结构，在 20 世纪 70 年代非常流行，C 和 UNIX 就是在这种体系结构下首次开发出来的。

period（**句点**）——点字符的另一个名字。

Peripheral Interchange Program（PIP）——外设交换程序，一些老的操作系统使用它转换文件和设备格式。

pointer type（**指针类型**）——表示函数或者数据对象类型的地址的数据对象类型。

portability（**可移植性**）——移到另一个环境下比为这个环境重新编写代码更经济。

POSIX——IEEE 1003 标准操作系统接口，它建立在 UNIX 为应用程序提供的系统服务的基础上。

precision（**精度**）——可以确切表示的值的数目，通常用位或十进制数字表示（它们指明了可以确切表示的值数目的对数）。

predicate（**谓词**）——产生一个二进制结果的表达式，通常用非零表示真，零表示假。

preprocessor（**预处理器**）—— C 翻译器的一部分，处理面向文本的指令和宏调用。

primitive（**原语**）——执行关键服务的接口函数，经常是那些不能通过其他方式实现的函数。

print function（**打印函数**）——在格式串的控制下把编码值转换为文本的函数。

printable（**可打印的**）——得到一个有意义的结果，例如当写出到显示设备时，显示一个图形或者控制打印位置。

program（**程序**）——函数和数据对象的集合，计算机可以通过执行它来完成相应的 C 源文件集的语义目标。

program startup（**程序启动**）——程序执行过程中 `main` 函数调用之前的那段时间。

program termination（**程序终止**）——程序执行过程中 main 函数返回或者调用 exit 后的那段时间。

push back（**回退**）——把一个字符返回给输入流，这样它就是下一个要读取的字符。

punctuation（**标点符号**）——字母和数字之外的可打印字符，用作分割和划定字符序列。

R range error（**值域错误**）——用一个（或多个）实参值调用一个数学函数，生成的结果太大或者太小而不能表示为一个有穷值。

read function（**读函数**）——从一个流中获得输入的函数。

read-only（**只读**）——包含一个不能被修改的存储值。

recursion（**递归**）——在一个函数的生存期中调用这个函数。

representation（**表示**）——表示一种数据类型所用的位数，以及描述各种位模式的含义。

reserved name（**保留名字**）——只能用作某种特定目的的名字。

round（**舍入**）——根据某种规则使用缩减的精度来获得一种表示，例如就近舍入。

rvalue（**右值**）——指定某种类型的值的表达式（不一定要指定一个数据对象）。

S scan function（**扫描函数**）——在格式串的控制下把文本转换为编码值的函数。

scan set（**扫描集**）——扫描函数的转换说明符，指定了匹配字符的集合。

seek（**寻找**）——修改一个流的文件定位符来指定文件内的一个给定的字符位置。

semantics（**语义**）——一种语言中某个有效的记号序列的含义。

sequence point（**序列点**）——程序的某个地方，在这个地方，存储数据对象中的值处于一个已知的状态。

side effect（**副作用**）——执行一个表达式时，存储在一个数据对象中的值或者一个文件的状态发生了改变。

signal（**信号**）——在程序执行过程中要求得到立即关注的事件。

signal handler（**信号处理程序**）——发生一个信号时执行的程序。

signed integer（**有符号整数**）——可以表示正值和负值的整数类型。

signed-magnitude arithmetic（**有符号数值算术**）——一种二进制编码形式，这

种编码形式下，一个数的相反数只取它的符号位的补。

significance loss（**有效值丢失**）——浮点加法或减法中由于高位缺失导致的有效精度的缩减。

source file（**源文件**）—— C 翻译器可以翻译为一个目标模块的文本文件。

space（**空格**）——占据一个打印位置但不显示任何图形的字符。

stack（**栈**）——具有后进先出原则的列表。

stack frame（**栈帧**）——函数被调用时为调用栈分配的数据。

Standard C（**标准 C**）—— ANSI/ISO C 标准定义的 C 程序设计语言。

Standard C library（**C 标准库**）—— C 标准定义的、可以被任意托管 C 程序使用的函数、数据对象和头文件的集合。

standard header（**标准头文件**）—— C 标准定义的 15 个头文件之一。

state table（**状态表**）——定义了一个有穷状态机的行为的数组。

statement（**语句**）——函数的可执行的组成部分，指定了某种动作，例如对一个表达式求值或者修改控制流。

static storage（**静态存储空间**）——一种数据对象，它们的生存周期从程序启动一直到程序终止，在程序启动之前进行初始化。

store（**存储**）——用一个新值取代存储在某个数据对象中的值。

stream（**流**）——维护一个打开的文件的一系列读、写和文件定位请求的状态的数据对象。

string（**串**）——存储在一个数组中的字符序列，这个数组的最后一个元素（下标最高的元素）是一个空字符。

string literal（**串字面量**）—— C 源文件中双引号之间的记号，例如 "abc"，它指定一个初始化为指定字符序列末尾再加上一个空字符的 *char* 类型只读数组。

structure type（**结构类型**）——由不同类型的数据对象成员组成的数据对象类型。

stub（**存根**）——函数的低级形式，用于测试或者作为函数正确实现之前的形式。

Sun UNIX——为 Sun 工作站提供的 UNIX 操作系统的一个版本。

synchronous signal（**同步信号**）——在程序执行过程中发生的重要事件，例如除零。

synonym（**同义词**）——声明一种类型的另一种方式，不过它等价于原始类型。

syntax（**语法**）——对语言中有效的记号序列的语法限制。

System/370——20 世纪 60 年代 IBM 开发的一种至今仍广泛使用的计算机体系结构，尤其适用于特别大的应用程序。

system call（**系统调用**）——系统服务的另一个叫法。

system service（**系统服务**）——请求操作系统执行某种服务，例如向一个设备写入数据或者获得当前时间。

T

text（**文本**）——适合写入到一个显示设备（可以被人们识别）的字符序列。

text stream（**文本流**）——包含文本的流。

thousands separator（**千位分隔符**）——用来对小数点左边的数进行分组的字符（不一定是 3 个字符一组）。

thread of control（**控制线程**）——一个程序实体的执行。

time zone（**时区**）——地球上的一个区域，这个区域的本地时间和 UTC 有固定的差值。

token（**记号**）——在高级语法中可以作为一个元素对待的字符序列。

translation table（**转换表**）——指明了两种编码方式之间对应关系的数组。

translation unit（**翻译单元**）——一个 C 源文件加上 `#include` 指令包含的所有文件，不包括条件编译语句跳过的源文件行。

translator（**翻译器**）——将翻译单元转换成可执行形式的程序。

truncate（**截断**）——向零舍入。

Turbo C++——Borland 公司为 PC 兼容机开发的 ANSI C（和更新的语言 C++）实现。

two's-complement arithmetic（**2 的补码算术**）——一种二进制编码方式，这种编码方式下，一个数的相反数的编码是按位取反加一。

type（**类型**）——一个值的属性（确定它的表示及可以对它执行什么样的操作）或一个函数的属性（确定它可以接受什么样的参数及有什么样的返回值）。

type definition（**类型定义**）——对某种类型进行命名的声明。

U

underflow（**下溢**）——一个值的计算结果太小而不能作为指定浮点类型表示。

union type（**联合类型**）——由可选的多种数据对象成员组成的数据对象，每次只能表示其中的一种成员类型。

UNIX——AT&T 贝尔实验室在 20 世纪 70 年代开发的一个独立于机器的操作

系统，C 语言的第一个宿主环境。

unsafe macro（**不安全的宏**）——可以对一个或者多个参数进行多次求值的宏，因此对于那些具有副作用的参数，这些宏可能产生意想不到的结果。

unsigned integer（**无符号整数**）——只能表示零和某个正数上限之间的值的整数类型。

ULTRIX——由 DEC 封装和支持的适用于 VAX 体系结构的 UNIX 版本。

uppercase letter（**大写字母**）——英语字母表中 A~Z 这 26 个字母中的一个，还可能加上 "C" 区域设置之外其他的附加字符。

Universal Time Coordinated（UTC）——协调世界时，GMT 的现代称呼。

V variable（**变量**）——数据对象的旧称。

variable argument list（**可变参数表**）——函数的参数列表，它可以接受声明的最后参数之外的附加参数。

VAX——继 DEC PDP-11 之后开发的 DEC 计算机体系结构，C 和 UNIX 仍然在这种体系结构下广泛使用。

void type（**void 类型**）——没有任何表示方法和值的类型。

volatile type（**volatile 类型**）——为那些可能被多个控制线程访问的数据对象设计的限定类型。

W WG14——ISO 授权负责 C 标准化的委员会。

white-space（**空白**）——一个或者多个空格字符的序列，可以和其他字符，例如水平制表符等混合。

wide character（**宽字节字符**）——用来表示大字符集的 wchar_t 类型的编码值。

width（**宽度**）——格式中转换说明的一部分，它部分控制要传送的字符数。

writable（**可写的**）——值可以改变，和只读相对应。

write function（**写函数**）——把输出传送到一个流的函数。

X X3J11——ANSI 授权开发原始的 C 标准的委员会。

Z zero fixup（**零修正**）——用零代替发生下溢的浮点值。

"绝对一流的著作，C程序员的饕餮大餐！"

——C/C++用户协会

The Standard C Library

C标准库

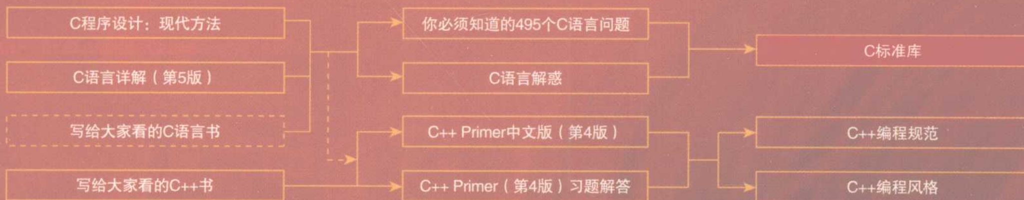
本书是由世界级C语言专家编写的C标准库经典著作。英文版已经重印十多次，影响了几代程序员。

本书结合C标准的相关部分，精辟地讲述了每一个库函数的使用方法和实现细节，这正是一个真正的C程序员所必须掌握的。更重要的是，书中给出了实现和测试这些函数的完整源代码，可以让你更深入地学习C语言。不仅如此，本书还讨论了一些即使是最有经验的C程序员通常也不熟悉的知识，比如国际化和独立于区域设置的程序的编写、与构建库相关的概念和设计思想。



P. J. Plauger 世界著名的软件技术专家，曾任ISO C标准委员会主席，*C/C++ User's Journal*主编，现任ISO C++标准委员会主席。他是C/C++标准库开发领域的大师，所开发的Dinkumware标准库应用广泛。

图灵C/C++图书阅读路线图





www.pearsonhighered.com

本书相关信息请访问：图灵网站 http://www.turingbook.com

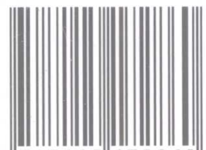
读者/作者热线：(010)51095186

反馈/投稿/推荐信箱：contact@turingbook.com

分类建议 计算机/程序设计/C

人民邮电出版社网址：www.ptpress.com.cn





ISBN 978-7-115-17286-0/TP

定价：79.00 元